VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PARALELNÍ GENETICKÝ ALGORITMUS

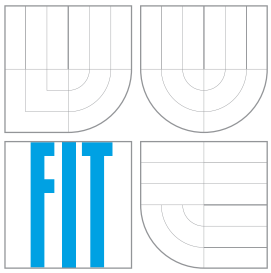
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN TRUPL

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PARALELNÍ GENETICKÝ ALGORITMUS

PARALLEL GENETIC ALGORITHM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN TRUPL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ

BRNO 2008

Abstrakt

Práce popisuje návrh a implementaci různých evolučních algoritmů, vylepšených tak, aby mohly využívat výhod paralelismu na víceprocesorových systémech, a zároveň umožňovaly, aby výpočet probíhal na více počítačích v počítačové síti. Algoritmy jsou určeny k hledání globálního extrému funkce několika proměnných. Jsou nastíněny různé zajímavé optimalizační problémy a možnosti jejich řešení právě pomocí evolučních algoritmů. V práci je rovněž rozebíráno použití knihovny rozhraní MPI (message passing interface) a OpenMP, v rozsahu nutném pro pochopení problematiky implementace paralelních evolučních algoritmů.

Klíčová slova

genetický algoritmus, evoluční algoritmus, Message Passing Interface, MPI, OpenMP, globální extrém, minimum, maximum, cenová funkce, fitness funkce, paralelizace, paralelní, neuronová sít, inverzní fraktální problém, koeficienty fourierovy řady, standardní genetický algoritmus, SGA, diferenciální evoluce, DE, samoorganizační migrační algoritmus, SOMA, stochastický horolezecký algoritmus, SHA

Abstract

The thesis describes design and implementation of various evolutionary algorithms, which were enhanced to use the advantages of parallelism on the multiprocessor systems along with ability to run the computation on different machines in a computer network. The purpose of these algorithms is to find the global extreme of function of n variables. In the thesis, there are demonstrated various optimization problems, and their effective solution with the help of evolutionary algorithms. There are also described interface libraries MPI(Message Passing Interface) and OpenMP, in the extent needed to understand the problematic of parallel evolutionary algorithms.

Keywords

evolutionary algorithm, genetic algorithm, Message Passing Interface, MPI, OpenMP, global extreme, minimum, maximum, cost function, fitness function, parallelisation, parallel, neural network, inverse fractal problem, coeficients, fourier, queue, standard genetic algorithm, SGA, differential evolution, DE, self-organizing migrating algorithm, SOMA, stochastic hill climbing, SHC

Citace

Jan Trupl: Paralelní genetický algoritmus, bakalářská práce, Brno, FIT VUT v Brně, 2008

Paralelní genetický algoritmus

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Trupl
13. května 2008

Obsah

Kapitola 1

Úvod

Jak v inženýrské, tak i ve vědecké praxi se často setkáváme s problémem, kdy hledáme optimální parametry nějakého systému, tak, aby výstup tohoto systému co nejvíce odpovídal naší potřebě. Je-li možné vytvořit matematický model takovéhoto systému, a funkci, jenž na základě číselně zadaných parametrů tohoto systému je schopna určit jeho kvalitu jakožto míru splnění našich požadavků, můžeme se pokusit použít různé evoluční techniky k nalezení optimálního řešení. Ať již potřebujeme najít ideální rozměry pece, tvar křídel letadla, váhové koeficienty neuronů v umělých neuronových sítích, či pouhé řešení nějakého matematického problému, mají nám evoluční techniky co nabídnout. Co ale jsou tyto evoluční techniky? Zjednodušeně řečeno se jedná o různé algoritmy, které pracují nad množinou jistého počtu tzv. jedinců, kteří představují vždy jedno konkrétní řešení daného problému. Tento soubor jedinců se nazývá populace, a každému jedinci je možno přiřadit číslo, které vyjadřuje jeho kvalitu. Evoluční algoritmy dovedou brát existující jedince z populace, a na základě určitých pravidel vytvářet jedince nové. Tito jsou pak zařazováni do nové generace řešení. Toto proběhne mnohokrát, dokud nedojde ke splnění nějaké podmínky zastavení evoluce. Takovýto postup obvykle vede k tomu, že v populaci začnou převládat jedinci vhodní, mezitím co jedinci méně vhodní jsou vylučováni, takže začnou tvořit v populaci minoritu.

Zdá se, že již ze svého samotného principu hodně operací prováděných během evoluce „in silico“ (v počítači) je na sobě vzájemně nezávislých. Co kdyby pracovaly souběžně? Pokud by tak bylo učiněno, zajisté by došlo k určitému urychlení, ale stojí režie s tímto spojená za to? Právě odpověď na tuto se snaží nalézt tato práce.

V následujících kapitolách budou popsány nejen možnosti, jak provádět výpočty v rámci multiprocesorového systému se sdílenou pamětí, ale rovněž, jak výpočet rozložit mezi vícero strojů, jenž se dorozumívají v rámci běžné počítačové sítě. Budou popsány problémy, jenž poslouží k demonstraci efektivity zvolených řešení, použité evoluční algoritmy, a návrh a implementace zvolených řešení. Na závěr budou shrnuty výsledky a grafy, porovnávající efektivitu zvolených řešení jak mezi sebou, tak vůči řešení sekvenčnímu. Rovněž budou naznačeny možnosti dalšího praktického rozšíření a pokračování výsledků této práce.

Kapitola 2

Evoluční algoritmy

2.1 Typy evolučních algoritmů

Rozhodneme-li se použít nějaký evoluční algoritmus, zjistíme, že existuje mnoho typů, z nichž si můžeme vybrat. Vybírat je třeba s rozvahou, protože pro různé úlohy mohou existovat takové evoluční algoritmy, které jsou více vhodné než jiné, či vůbec není nutno je používat - evoluční algoritmy mají význam především pro problémy NP-úplné nebo s $n!$ složitostí, a třeba takové polynomiální úlohy je lepší řešit klasicky. Na druhou stranu je pravda, že evoluční algoritmy dovedou řešit řadu jistým způsobem formalizovaných úloh, aniž by je ovlivňoval skutečný význam dané úlohy, takže úlohu stačí vhodně formalizovat a předat evolučnímu algoritmu, aniž by bylo nutné dále implementovat nějaký konkrétní algoritmus. Je jedno, jestli se pomocí nich řeší, jak bude popsán stavový automat popisující pohyb virtuálního mravence ve virtuálním bludišti, či váhové koeficienty neuronové sítě. Jsou do jisté míry velice univerzální. Každopádně volba správného algoritmu může mít kritický význam na jeho dobu běhu, zvláště, existuje-li potřeba hledat alespoň suboptimální řešení v pravidelných intervalech.

2.2 Formalizace problémů

2.2.1 Vhodné typy úloh

Chceme-li řešit nějakou optimalizační úlohu pomocí evolučního algoritmu, měli bychom nejprve zvážit, zdali neexistuje řešení jednodušší. Je evidentní, že pro hledání extrému funkce jedné proměnné nemusí být použití takovéhoto algoritmu nejefektivnějším řešením, vzpomeňme na pověstný kanón na vrabce. Úlohy, jenž jsou vhodnými adepty pro řešení pomocí evolučního algoritmu, se obvykle vyznačují značným počtem parametrů, tj. představují hledání extrému funkce v n-rozměrném prostoru, kde n může být i poměrně velké číslo. Mohou být také multimodální, což znamená, že existuje více přibližně stejných globálních extrémů.

2.2.2 Formalizace optimalizační úlohy

Problém, který chceme řešit, je nutno nějakým způsobem matematicky popsat, popřípadě zakódovat do formátu vhodného pro daný evoluční algoritmus.

Jedinec a specimen

Při hledání extrému funkce více proměnných je stav řešení úlohy popsán jako uspořádaná n-tice čísel $(x_1, x_2, \cdots, x_n)$, které se v rámci terminologie evoluční algoritmů říká *jedinec*. Definiční obor jednotlivých x_i je dán pomocí takzvaného *specimenu*, což je v podstatě předpis, říkající, jakých hodnot mohou souřadnice jedince nabývat. Specimen je tedy předpis ve tvaru $S = ([l_1, h_1], [l_2, h_2], \cdots, [l_n, h_n])$, kde každá usořádaná dvojice $[l_i, h_i], i \in \{1, \cdots, n\}$ určuje přípustný definiční obor parametru x_i jedince.

Cenová funkce

Aby mohla být porovnána kvalita jednotlivých jedinců, je třeba definovat tzv. cenovou funkci, což je v podstatě zobrazení bodu n-rozměrného prostoru na reálné číslo:

$$f(x_1, x_2, \cdots, x_n) \rightarrow R$$

Způsob, jakým toho tato funkce dosáhne, je pochopitelně závislý na konkrétní úloze. Může se jednat třeba jen o jednoduchý výpočet nějakého vzorce, ale také to může znamenat kompletní simulaci nějakého složitého modelu.

2.2.3 Standardní genetický algoritmus

Standardní genetický algoritmus je snad vůbec první evoluční algoritmus, který byl vymyšlen. Jeho základem je tzv. „ruletové kolo“, což je selekční strategie, která funguje zhruba tak, že podle ohodnocení jedinců se do velkého pole vygenerují jejich indexy, a to tak, aby se nejlepší jedinec vyskytoval nejčastěji, průměrný méně často, a podprůměrný málo, a nebo vůbec. Potom se generuje náhodné číslo takové, aby mohlo toto pole indexovat, a vezme se index, který se nachází na takto zvolené pozici v tom poli, a ze staré populace se tímto indexem indexovaný jedinec překopíruje do nové. Lze snadno nahlédnout, že takovýto postup časem povede k převládání jedinců z lepšími vlastnostmi. Takováto selekce však sama o sobě nestačí, protože nepřináší do populace tzv. „evoluční novinky“. Nástroji vhodnými pro zvýšení diverzibility populace jsou *mutace* a *křížení*. Mutace spočívá v tom, že se pro každý bit jedince náhodně rozhodne, jestli bude invertován, nebo ne. Křížení pak znamená, že se z populace vybere náhodný sudý počet jedinců, a pak vždy po sobě jdoucí dvojice se skříží. Obvykle se provádí křížení jednobodové, tj. vybere se náhodné místo, kde se jedinci rozpojí, a všechna data od začátku vektoru jedince až po toto místo si jedinci prohodí(nebo od tohoto místa až po konec jedince) - jedná se tedy o klasický *crossover*, tak, jak ho známe i z biologie. Je nutno ještě dodat, že v současné době se místo ruletové selekce často používá tzv. *turnajové selekce*, jenž spočívá v tom, že se z populace náhodně vyberou dvojice, mezi nimiž se udělá „zápas“(porovnání kvality), a ti, jenž projdou, jsou zařazeni do nového kola turnaje (lze vytvořit mnohem více dvojic, než je velikost populace - z populace o velikosti N celkem až $\binom{N}{2}$ dvojic). Turnaj pokračuje tak dlouho, dokud je počet „soutěžících“ vetší než velikost cílové populace. Více se o dané problematice lze dozvědět v [4]

2.2.4 Diferenciální evoluce

Historie

Diferenciální evoluce vznikla jako důsledek snahy Kennetha V. Price vyřešit Chebychevův polynomiální aproximační problém, který mu předložil Rainer Storn. Původ diferenciální

evoluce spočívá v tzv. genetickém žíhání (genetic annealing)(Price, 1994), které bylo vyvinuto rovněž K. V. Pricem, a které pro reprezentaci problému používalo binární kódování. Následně mu bylo navrženo, aby pomocí tohoto algoritmu pokusil řešit i složitější úlohy, což vedlo k tomu, že K. V. Price začal měnit genetické žíhání z reprezentace binární do dekadické a úměrně tomu operace logické na vektorové. Tyto změny udělaly z genetického žíhání algoritmus vhodný pro numerickou optimalizaci. Jakmile byly provedeny tyto změny, byla K. V .Pricem velmi rychle objevena tzv. diferenciální mutace (differential mutation) (Price, 1999), která spočívá v tom, že generuje zkušební řešení přičtením diference dvou náhodně vybraných vektorů (jedinců) ke třetímu jedinci v populaci. Poté byla zkombinována diferenciální mutace a metody selekce z genetického žíhání, a tak vzniklo to, co dnes označujeme jako první verzi diferenciální evoluce. Vzhledem k tomu, že principy žíhání aplikované v genetickém algoritmu se ukázaly být nadbytečnými, byly z diferenciální evoluce vypuštěny. Zároveň s tím R. Storn navrhl vytváření populace potomků ve zvláštní populaci, jenž se účastní soupeření o místo v nové populaci až po naplnění této zvláštní populace, a zkrátil původní dlouhý název na stručný název diferenciální evoluce. Po čase vznikla třetí verze diferenciální evoluce, jakožto důsledek snahy napravit nedostatečnost původních verzí z hlediska schopnosti řešit složitější optimalizační úlohy. Více se lze o diferenciální evoluci dozvědět na stránkách [5] a [6], kde je k dispozici mnoho zdrojových textů včetně on-line demonstrací a PDF souboru s vysvětlením problematiky.

Popis algoritmu

Diferenciální evoluce pracuje nad populací jedinců, kde každý jedinec je reprezentován jako vektor obvykle reálných čísel. Evoluce probíhá generačně, v každém kroku evoluce je za pomoci mezigenerace mutantů vytvořena nová generace, nahrazující starou. Nový jedinec je vytvořen tak, že ze staré generace se vyberou čtyři různí jedinci, z nichž jeden je ten nahrazovaný, a ostatní tři jsou vybráni náhodně. Pak se provede vektorový rozdíl prvních dvou náhodně vybraných jedinců, a nově vzniklý vektor, tzv. váhový diferenční vektor, se vynásobí mutační konstantou, označovanou velkým písmenem F. Třetí z náhodně vybraných vektorů, zvaný též bázový vektor, se sečte se vzniklým diferenčním váhovým vektorem, čímž vznikne nový jedinec, který se zařadí do populace mutantů na místo se stejným indexem, jako je jedinec nahrazovaný. Potom se vytvoří tzv. *testovací jedinec*, a to tak, že se postupně berou jednotlivé složky jedince nahrazovaného a jedince z populace mutantů, a pro každou takovouto dvojici se vygeneruje náhodné číslo v intervalu $\langle 0, 1 \rangle$, a porovná se s prahem křížením CR, a je-li menší, bere se jako nová složka vektoru jedince odpovídající složka vektoru jedince z populace mutované, jinak se bere odpovídající složka původního jedince. Nyní se za pomoci cenové funkce vypočítá kvalita takto nově vzniklého testovacího jedince, a je-li lepší, než kvalita odpovídajícího jedince v generaci původní, je do nové generace na toto místo zařazen testovací jedinec, jinak je do nové generace zařazen jedinec původní. Z výše uvedeného popisu je zřejmé, že aby algoritmus mohl pracovat, musí mít populace alespoň čtyři jedince. Podrobnější informace o algoritmu lze dohledat na [5] a nebo česky v [7].

Samoorganizační migrační algoritmus

Samoorganizační migrační algoritmus je evoluční algoritmus, který existuje od roku 1999, a postupem času byl vylepšován. Mezi evoluční algoritmy může být řazen i přesto, že v podstatě nevytváří novou populaci, ale jednotliví jedinci migrují po hyperploše prohledávaného prostoru. Myšlenka algoritmu je zhruba taková, že v prohledávaném prostoru jsou různě

kvalitní jedinci, přičemž na „kvalitu“ se můžeme dívat jako na množství potravy, které se na daném místě nachází. Jedinci se vždy rozhodnou cestovat směrem k jedinci s nejlepším ohodnocením, přičemž během své cesty prohledávají prostor, zdali nenaleznou řešení lepší. Důležitou vlastností tohoto „cestování“ je, že se odehrává $N - k$ rozměrném prostoru, tj. je vyloučen náhodný počet dimenzí. Jedinci jsou při cestě tedy náhodně odkloněni, čímž dojde k důkladnějšímu prohledání domény problému. Bez této vlastnosti by se tento algoritmus hodil pouze k hledání lokálních minim. Existují také různé jiné strategie pohybu jedinců, než k nejlepšímu. Může se také zkoušet migrovat k náhodnému, či všichni ke všem, což je důkladnější, ale také výpočetně náročnější. Více se o daném algoritmu dá dozvědět v [7].

2.2.5 Stochastický horolezecký algoritmus

Stochastický horolezecký algoritmus(SHA) je jednoduchý algoritmus, který je možná docela odvážné nazvat evolučním. Každopádně může rovněž pracovat nad populací jedinců, a pro každého z nich vygenerovat náhodný směr, o který se v dané generaci posune, pokud je ohodnocení nového místa lepší než jeho aktuální. Do této práce byl zvolen pro svoji jednoduchost a také v domnění, že poslouží jako referenční algoritmus pro zrychlení pomocí pomocí paralelizace. Skutečně, SHA lze paralelizovat skoro absolutně, protože neexistují žádné vzájemné závislosti mezi jedinci populace. Nevýhodou je, že jedinci, kteří dosáhli lokálního extrému, se přestanou vyvíjet. Proto byl algoritmus vylepšen tak, že pokud ve *2x dimenze řešeného problému* generacích nedojde ke změně jedince, je jedinec nahrazen náhodně vygenerovaným jiným jedincem. Nejlepší nalezený jedinec je samozřejmě uchováván. Počet generací *2x dimenze řešeného problému* bylo zvoleno na základě intuitivní představy, že ve vícerozměrném prostoru je nutno náhodně vykročit do dvakrát tolika směrů, než kolik je počet dimenzí (jednou dopředu, jednou dozadu pro každou dimenzi). Výše uvedený popis SHA vychází z implementace intuitivně provedené v rámci této práce.

Kapitola 3

Metody paralelizace výpočtů

Paralelizace výpočtů znamená, že výpočet je prováděn na více počítačích(stanicích), či procesorech. Toto může být kombinováno, a to tak, že výpočet probíhá na počítačích propojených v síti, a každý z těchto počítačů má více procesorů, mezi něž jsou distribuovány procesy. To je samozřejmě nejideálnější situace, jelikož prostředky jsou využívány optimálně. Stanice s vícejádrovým procesorem se z hlediska systému tváří stejně, jako stanice mající skutečně více procesorů. Mohou existovat i systémy, mající několik vícejádrových procesorů.

Pro paralelizaci na více stanicích je obvykle využíváno prostředí s distribuovanou pamětí, tj. paměť není sdílena mezi jednotlivými procesy, a je nutné, aby data, jenž mají být společná, byla pravidelně aktualizována např. zasíláním po síti. Tuto problematiku řeší rozhraní MPI (Message Passing Interface), ale čistě teoreticky nic nebrání tomu, aby programátor implementoval toto chování pomocí standardních funkcí systému pro práci se sítí.

Naopak na lokálních strojích běží většinou více vláken. Vlákna sdílejí paměť, a můžou nastat konflikty v přístupu k paměťovým místům (jedno vlákno zapisuje na stejné místo, z něhož se ve stejnou chvíli pokouší číst alespoň jedno jiné vlákno). V operačních systémech Unixového typu bývají vlákna podporována pomocí knihovny POSIX threads (pthreads), existují ale také kompilátory, které problematiku vláken řeší pomocí OpenMP.

3.1 Prostředí se sdílenou pamětí

3.1.1 OpenMP

OpenMP je efektivní způsob, jak rozšířit již existující program tak, aby některé části programu běžely paralelně ve více vláknech. Jedná se o rozšíření kompilátorů jazyků C/C++ a Fortran, které je nutno zapnout pomocí parametru kompilace, jenž bývá popsán v dokumentaci daného kompilátoru. Programátor zapisuje na určitá místa programu speciální pragma direktivy, které, pokud jsou rozpoznány, způsobí, že přeložený program na daném místě běží ve více vláknech. Výhoda tohoto řešení je zřejmá: pokud je program kompilován v prostředí, jenž OpenMP nepodporuje, přeloží se bez jeho podpory. Další výhodou OpenMP je, že využívá vláken namísto procesů „těžké váhy“, která startují mnohem rychleji a jsou méně náročná na prostředky operačního systému. Samozřejmě, že mohou existovat implementace, jenž využívají normální procesy systému (funkce fork()), ale význam mají především takové, jenž používají např. pthreads, Win32Threads, či obdobné prostředky. Z toho také plyne další výhoda OpenMP, kterou je nezávislost na konkrétní platformě.

Úvod do OpenMP

Jak již bylo naznačeno výše, program využívající možností OpenMP běží na některých svých místech ve více vláknech. Každé vlákno je nezávislý řídící agent, který pracuje uvnitř stejného adresového prostoru (AP) jako původní sekvenční program. Každé vlákno OpenMP má vlastní zásobník, paměť, registry, svoje pořadové číslo (index), stav provádění, ukozovátko instrukcí (IP) a zásobníku (SP). Správu vláken provádí jádro OS nebo uživatel pomocí knihovních funkcí. Program OpenMP se začne vykonávat jedním vláknem (master thread s indexem 0) a teprve až dojde k paralelnímu příkazu, vytvoří tým potřebných vláken. Členové týmu provádí příkazy paralelně a synchronizují se na bariéře. Program se může při běhu větvit a zase spojit mnohokrát (model fork-join). Je-li více vláken než procesorů, pak se vlákna mapují na procesory dynamicky. OpenMP tvoří:

- **direktivy,**
- **knihovní funkce** (např. `omp_set_num_threads()`, `omp_get_thread_num()`, `omp_get_num_threads()`, ...)
- **proměnné prostředí** (`OMP_NUM_THREADS`, ...)

Pokud kompilátor podporuje OpenMP, pak `#pragma omp` se bere jako direktiva OpenMP a všechny příkazy, jenž mají být vykonány pouze v paralelní verzi algoritmu je možno zapsat mezi `#ifdef _OPENMP` a `#endif`.

Předcházející text vyhází a cituje z 3. kapitoly v [2].

3.1.2 Příklad jednoduché aplikace v OpenMP

Jakožto nenásilný úvod do OpenMP je uvedena a rozebrána jednoduchá aplikace provádějící nějaký blíže nedefinovaný výpočet:

```
#define ARRAY_SIZE(X) (sizeof(X)/sizeof(X[i]))

int main()
{
  int data[10] = {0,1,2,3,4,5,6,7,8,9}, i;
  int data2[10] = {9,8,7,6,5,4,3,2,1,0};

  /* 1. */
  #pragma omp parallel for default(none) shared(data) private(i)
  for (i=0; i<ARRAY_SIZE(data); i++)
    data[i] += 2;

  /* 2. */
  #pragma omp parallel
  {
    /* 3. */
    #pragma omp for
    for (i=0; i<10; i++)
      data[i] += i;
```

```
    /* 4. */
    #pragma omp single
    data[0] += 1;
  }

  /* 5. */
  #pragma omp parallel for private(i) shared(data) nowait
  for (i=0; i<ARRAY_SIZE(data); i++)
    data[i] <<= 1;

  #pragma omp parallel for private(i) shared(data2)
  for (i=0; i<ARRAY_SIZE(data2); i++)
   data2[i] += data2[i] * data2[i] + (1<<i);

  return 0;
}
```

1. Paralelní smyčka for:
Následující cyklus for bude vykonán paralelně, a to tak, že se práce rozdělí mezi jednotlivá dostupná vlákna. Direktiva *shared(list)* obsahuje seznam proměnných, jenž budou sdíleny mezi vlákny. Změny provedené na proměnných v tomto seznamu budou zachovány i po skončení paralelní části programu. Direktiva *private(list)* obsahuje seznam proměnných, které má mít každé vlákno samostatně, tj. nejsou sdíleny. Direktiva *default(none)* znamená, že všechny proměnné použité ve smyčce musí být uvedeny buď na seznamu proměnných private, nebo na seznamu shared(list).
2. Paralelní sekce: všechny příkazy v paralelní sekci jsou vykonávány paralelně všemi vlákny, není-li určeno jinak.
3. Cyklus for uvnitř paralelní sekce: cyklus je rozdělen mezi více vláken, takže v každém vláknu proběhne jenom část smyčky, čímž dojde ke zkrácení celkové doby strávené ve smyčce.
4. Direktiva single: příkaz je proveden jenom jedenkrát, a to v prvním vláknu, jenž k němu dojde.
5. Paralelní smyčka bez implicitní bariéry a na ní navazující smyčka s bariérou: po provedení první smyčky nejsou vlákna synchronizována na bariéře, a okamžitě začnou vykonávat kód druhé smyčky. Za druhou smyčkou for je implicitní bariéra, na níž se vlákna synchronizují.

3.2 Prostředí s distribuovanou pamětí

3.2.1 Message Passing Interface - MPI

MPI je rozhraní pro podporu paralelního programování zasíláním zpráv (MP, Message Passing), jenž je obtížnější než programování se sdílenými proměnnými, protože komunikace musí programátor zapsat explicitně. Jedná se o standard, který není programovacím jazykem, ale sadou knihovních funkcí volaných ze sekvenčního programu. Tím je také zajištěna jistá přenositelnost programů [2].

Základní pojmy

Aplikace MPI je tvořena n procesy na 1 až n procesorech (MIMD nebo SPMD). Základem je párová komunikace (point-to-point), zprávy se nepředbíhají, z čehož plyne deterministické provedení algoritmů. MPI na nejvyšší úrovní definuje pojem komunikační domény - tzv. *komunikátor*. Komunikátor je skupina procesů a komunikačních kanálů mezi nimi, kde každý proces má svůj index, tzv. *rank*. Implicitní komunikátor je `MPI_COMM_WORLD`, a v rámci této práce nebudou jiné komunikátory použity. Více o této problematice viz. [2]. V každém programu jsou vždy přítomny funkce `MPI_Init()` a `MPI_Finalize()`, kde první slouží k inicializaci a poslední k ukončení práce s MPI. Mezi další významné funkce patří funkce pro zjištění počtu procesů ve skupině `MPI_Comm_size(MyWorld,&group_size)` a pro zjištění svého indexu ve skupině `MPI_Comm_rank(MyWorld,&my_rank)`.

3.2.2 Komunikace dvou procesů v rámci komunikátoru

Pro komunikaci mezi dvěma procesy je možno použít funkcí `MPI_Send` a `MPI_Recv`. Existují také tzv. neblokující verze `MPI_ISend` a `MPI_IRecv`. Pro potrvzení odeslání v případě neblokující verze slouží funkce `MPI_Wait`. Příklad:

```
...
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
if (myrank == 0) {
  ...
  int send_data = 12345;
  MPI_Isend(&send_data, 1, MPI_INT, msgtag, MPI_COMM_WORLD, &req1)
  ... // užitečné výpočty
  MPI_Wait(&req1,&status);
} else if (myrank == 1) {
  int recvd_data;
  MPI_Recv(&recvd_data, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, &status);
}
```

V příkladu jeden proces druhému posílá celé číslo typu int, a po neblokujícím odeslání počítá něco užitečného. Po dokončení svého výpočtu čeká na dokončení odeslání pomocí `MPI_Wait`. Druhý proces pomocí `MPI_Recv` čeká na příjem zprávy. Další informace o komunikaci více procesů lze zjistit opět v publikaci [2].

3.2.3 Kolektivní operace

Kolektivní operace v MPI jsou takové operace, kdy spolu najednou komunikují více než dva procesy. Jedná se tedy o výměnu dat různými procesy.

MPI_Allgather

Výměna dat mezi všemi procesy. Máme N procesů, z nichž každý chce odeslat data o velikosti k bajtů. Pak každý proces musí mít pole pro přijetí výsledků o velikosti $N * k$. Procesy si tedy vymění data každý s každým, a na konci operace mají všichni všechna data ostatních. Následuje příklad kódu s vysvětlivkami v komentářích.

```
int item_size = 8, grp_size; // velikost položky 8 bajtů
MPI_Comm_size(MPI_COMM_WORLD, &grp_size); // zjistíme počet MPI procesů

// buffer pro posílání
std::vector<unsigned char> send_buffer(item_size);

// buffer pro příjem
std::vector<unsigned char> recv_buffer(item_size * grp_size);

// kolektivní operace, po jejímž ukončení všechny procesy
// mají obsah send_bufferu ostatních procesů
// ve svém recv_bufferu
MPI_Allgather(
  &send_buffer[0],item_size,MPI_BYTE,
  &recv_buffer[0],item_size,MPI_BYTE,
  MPI_COMM_WORLD);
```

MPI_Allreduce

Tato funkce slouží k tomu, že se nad daty poslanými od všech procesů vykoná nějaká operace, jejíž výsledek je nakonec např. jedno číslo. Příklad:

```
// budeme redukovat ukončovací podmínku algoritmus
int stop = ..., stop_reduced;

// logické nebo
MPI_Allreduce(&stop,&stop_reduced,1,MPI_INT,MPI_LOR,MPI_COMM_WORLD);

// redukce ukončovací podmínky true => konec
if (stop_reduced) exit();
```

3.2.4 Synchronizace

K synchronizaci MPI procesů s komunikátorem `MPI_COMM_WORLD` slouží funkce `MPI_Barrier(MPI_COMM_WORLD)`. Ta způsobí, že všechny tyto procesy dojdou do tohoto místa, a dále pokračují až všechny společně.

3.2.5 Závěr

Problematika MPI je detailně rozebrána v publikaci [2], kde lze dohledat množství podrobnějších a přesnějších údajů. Mnoho informací lze také nalézt na WWW [1].

3.3 Teoretické zrychlení výpočtů

Chceme-li zjistit, k jakému zrychlení dojde při použití kombinace OpenMP + MPI, použijeme následující vzorec:

$$S = \frac{1}{A_{serial} + \frac{A_{bad}}{N_{threads}} + \frac{A_{good}}{N_{procs}}}$$

kde:

- A_{serial} značí podíl kódu, jenž není možno paralelizovat
- A_{bad} je podíl kódu, který se nevyplatí paralelizovat pomocí MPI
- $N_{threads}$ je počet vláken OpenMP na daných stanicích (předpokládá se všude stejný)
- A_{good} je podíl kódu, jenž se vyplatí paralelizovat pomocí MPI
- N_{procs} je počet strojů SMP, mezi něž bude výpočet rozdělen pomocí MPI
- S je předpokládané zrychlení oproti sekvenční verzi algoritmu

Máme-li např. v nějaké aplikace 5% kódu sekvenčního, 90% kódu perfektně paralelizovatelného a 5% se paralelizovat nevyplatí, a k dispozici je 16 2-procesorových SMP stanic, kde na každé poběží dvě vlákna, pak teoretické zryhclení vychází jako

$$S = \frac{1}{0,05 + \frac{0,05}{2} + \frac{0,9}{16}} = 7.6$$

Výše uvedený vzorec je zobecněněním ukázkového výpočtu uvedeného v [2] a vychází z Amdahlova zákona, který je uveden tamtéž.

Kapitola 4

Programátorský návrh

Bylo rozhodnuto, že celá demonstrace algoritmů bude provedena tak, že bude vytvořena knihovna, jenž umožní, aby s ní programátor mohl pohodlně řešit všelijaké optimalizační problémy. Z hlediska programátora by se mělo jednat o poměrně jednoduchou záležitost, a měly by před ním být skryty detaily paralelizace jím užívaných genetických algoritmů. Knihovna tudíž bude mít sadu již naprogramovaných a efektivně paralelizovaných evolučních algoritmů, jenž budou mít jednotné rozhraní, a budou tedy zaměnitelné.

4.1 Návrh paralelizace

V následující sekci bude popsán návrh paralelizace využívající prostředků OpenMP a MPI. Budou naznačeny postupy, které by měly vést k urychlení.

4.1.1 Návrh užití MPI

V této podkapitole jsou shrnuty návrhy paralelizace pomocí MPI.

Ostrovní model

Ostrovní model znamená, že existuje více populací, jenž se vyvíjejí relativně nezávisle na sobě. Samozřejmě, že v určitých časových okamžicích může docházet k výměně užitečné informace mezi populacemi.

Jako informace vhodná pro výměnu v tomto modelu(každý s každým) se jeví nejlepší jedinec, také by bylo možno posílat jedince náhodného. Nemá asi cenu vyměňovat si celé kusy populace, protože by se snadno mohlo stát, že cílová populace bude zcela nahrazena novou, složenou z příchozích.

Důležité je, že každá populace „ví“ o všech ostatních, a dohromady tak tvoří úplný graf. V podstatě běží několik genetických algoritmů najednou. Toto se přímo nabízí k paralelizaci, a dává nám možnost promyslet, jak paralelizovat. Protože tématem této práce je hledání extrémů funkce, nabízí se jako jedna z možností rozdělit prohledávaný vícerozměrný prostor do prostorů několika. Toho se dá nejsnáze dosáhnout rozdělením jedné dimenze podle počtu populací, jenž chceme, aby se vyvíjely nezávisle. Pravděpodobně nejvhodnější je rozdělit tu dimenzi, která má největší rozdíl mezi maximální a minimální přípustnou hodnotou, aby se minimalizovala pravděpodobnost, že nějaký jedinec poruší omezení na něj kladená, což by vedlo k náhodnému generování souřadnice nové, a to zase k přílišné náhodnosti použitého algoritmu.

Kruhový model

Kruhový model je metoda, kdy jsou všechny nezávisle se vyvíjející populace uspořádány do kruhové topologie, a sousední populace si mohou mezi sebou vyměnit určité procento jedinců. Zatížení sítě se tak udrží na rozumné míře, a nemělo by ani docházet k tomu, že přijde více jedinců, než je cílová populace schopna pojmout(předpoklad konstantního počtu jedinců v populaci), což by bylo neefektivní, protože by bylo zbytečné takové nevyužité jedince posílat. Jako možnost posílání jedinců se jeví poslat část populace procesu s indexem o jedna vyšší, a nahradit tuto poslanou část jedinci přijatými od procesu s indexem o jedna nižší.

Shrnutí

Následuje shrnutí návrhů optimalizace pomocí MPI.

- Nezávislá evoluce více populací, a pravidelné srovnání dosažených výsledků.
- Migrace nejlepšího jedince v pravidelných intervalech, populace si vymění nejlepší jedince.
- Migrace náhodného jedince mezi populacemi v pravidelných intervalech.
- Migrace části populace ke svým sousedům v rámci kruhového modelu.

4.1.2 Návrh užití OpenMP

Pro správně provedenou paralelizaci pomocí sdílené paměti je nutno se zamyslet, které části evolučního algoritmu jsou nejvíce výpočetně náročné, a jenž bude třeba optimalizovat. V každém genetickém či evolučním algoritmu dochází určitě k inicializaci, vyhodnocení populace, páření a tvorbě nové generace a testování na splnění ukončovacích podmínek. Z toho vše až na inicializaci se děje v cyklu, a ohodnocení jedinců nové generace může být součástí páření, pokud během něho dochází k ohodnocení různých zkušebních jedinců. Je zřejmé, že ohodnocování jedinců a také jejich páření, představují stěžejní a výpočetně nejvíce náročné operace. Proto to budou tyto, jenž budou přednostně paralelizovány. Protože až na inicializaci vše probíhá v cyklu, bude paralelizována celá smyčka evolučního algoritmu, a ty části, které nebudou paralelizovány, budou provedeny pouze hlavním vláknem. Tímto se sníží režie spojená s vytvářením a rušením vláken.

Paralelizace smyčky vyhodnocující jedince

Protože ohodnocování každého jedince pomocí cenové funkce jsou na sobě vzájemně nezávislá, představuje toto místo algoritmu ideální příležitost pro paralelizaci. Paralelizace bude tedy provedena tak, že cena jednotlivých jedinců bude vyhodnocována paralelně, a to pomocí paralelní for smyčky. Toto přinese zásadní urychlení především tehdy, je-li cenová funkce dostatečně výpočetně náročná. U malé populace s jednoduchou cenovou funkcí lze předpokládat, že může dojít spíš k prodloužení výpočtu v důsledku režie spojené s prováděním vláken.

Paralelizace generování nové populace

Při generování nové populace jsou podle určitých pravidel vytvářeni noví jedinci, přičemž jsou buď zároveň ohodnocování, a na základě porovnání s předkem zařazeni do nové populace, nebo jsou prostě jenom zařazeni, aniž by došlo k ohodnocení. Protože algoritmy

zde popisované vesměs patří mezi ty složitější, bude mít paralelizace generování nové populace opravdu význam. Například SOMA během generování nového jedince vyžaduje značné množství volání cenové funkce, a rovněž diferenciální evoluce potřebuje jedno ohodnocení zkušebního jedince. U SGA bude změřeno, zdali má paralelizace pozitivní efekt.

4.2 Návrh rozhraní objektů

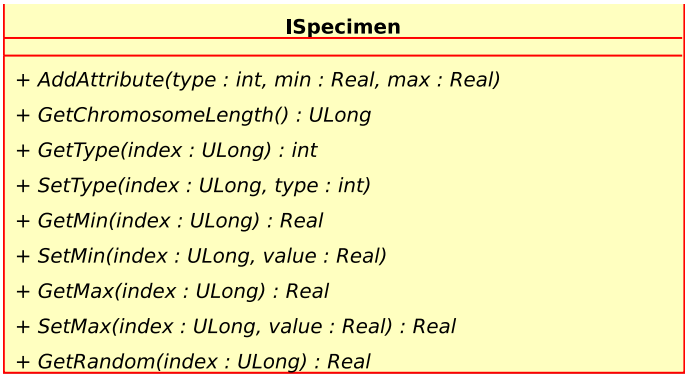
Knihovna je navržena tak, že existuje sada standardních rozhraní pro jednotlivé typy objektů (jedinec, cenová funkce, populace ...), a dále sada objektů, jenž z tohoto rozhraní dědí.

Jako kostra, jenž později posloužila pro základ implementace, byla navržena sada rozhraní, jenž zahrnuje hrubou funkcionalitu, potřebnou v každém evolučním algoritmu. Cílem bylo udělat rozhraní co nejobecnější, což jak se později ukázalo vedlo občas na méně efektivní implementaci, protože rozhaní fakticky znemožňuje používat přímý přístup do paměti, což je ale logické, protože účelem rozhraní je skrýt implementační detaily. Samotná implementace bude provedena jako sada objektů, jenž z těchto rozhraní dědí. Pokud spolu nějaké objekty budou navzájem komunikovat, bude se tak dít vždy za pomocí rozhraní jejich předků, čímž bude zvýšena flexibilita knihovny. Všechny třídy budou součástí jmenného prostoru `PGALib`.

Následuje výčet navržených rozhraní, nutno poznamenat, že detailnější informace lze dohledat pomocí dokumentace vygenerované programem doxygen nacházející se na přiloženém CD.

4.2.1 ISpecimen

Rozhraní *ISpecimen* slouží k určení hranic prohledávaného prostoru, čehož by mělo být využito při tvorbě nových jedinců. Pro každý rozměr umožňuje zadání a zjištění mezí a informace o definičním oboru. Definiční obor je z hlediska specimenu pouhé číslo, které ho identifikuje, a je na uživateli, aby tuto informaci dodal a případně použil při ohodnocování jedince. Minimální a maximální meze určují hranice daného rozměru. Rovněž pro daný rozměr by mělo být možno vygenerovat validní náhodné číslo.



Obrázek 4.1: Rozhraní ISpecimen

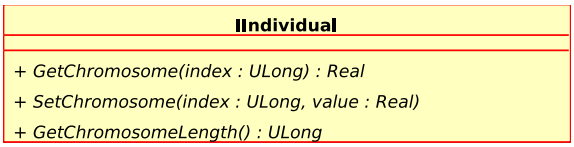
- `AddAttribute(type, min, max)` přidání rozměru s omezením $\langle min, max \rangle$

- `GetChromosomeLength()` zjištění dimenze prohledávaného prostoru
- `GetType(index)` a `SetType(index)` zjistí, či nastaví typ místa chromozomu
- `GetMin(index)` a `GetMax(index)` získá minimální, resp. maximální hodnotu pro daný index
- `SetMin(index)` a `SetMax(index)` nastaví minimální, resp. maximální hodnotu pro daný index
- `GetRandom(index)` vrátí platnou náhodnou hodnotu pro daný index

Index určuje místo na „chromozómu“, a nabývá hodnot `0` až `GetRandom()-1`.

4.2.2 IIndividual

Rozhraní *IIndividual* je rozhraní, které popisuje základní metody, které každý jedinec musí implementovat. Protože se zabýváme numerickou optimalizací, je jedinec v podstatě vektor reálných čísel. Proto jsou přítomny *setter* a *getter* pro „chromozóm“ jedince, a metoda pro zjištění délky jeho „chromozómu“. Způsob alokace paměti uchovávající potřebná data je takto ponechán na konkrétním implementujícím objektu. Drobnou nevýhodou tohoto návrhu je copy-konstruktor, který je nutno implementovat právě pomocí rozhraní tohoto objektu, což se může projevit na rychlosti kopírování jedince.



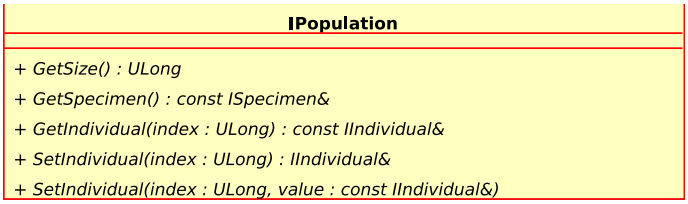
Obrázek 4.2: Rozhraní IIndividual

- `GetChromosome(index)` vrátí hodnotu chromozóm jedince na pozici *index*
- `SetChromosome(index,value)` nastaví hodnotu chromozómu jedince na pozici *index* na hodnotu *value*
- `GetChromosomeLength()` vrátí délku chromozómu jedince

4.2.3 IPopulation

IPopulation zahrnuje celou populaci jedinců, vygenerovaných podle stejného specimenu. Z návrhu rozhraní (viz. níže) je zřejmé, že chybí metoda nastavení velikosti populace. Předpokládá se, že toto bude řešit objekt, jenž toto rozhraní bude implementovat.

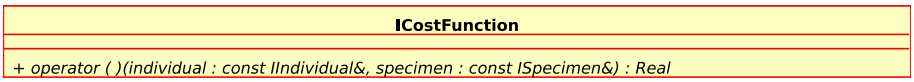
- `GetSize()` vrátí velikost (=počet jedinců) populace
- `GetSpecimen()` vrátí referenci na specimen, podle kterého je populace vygenerována
- `GetIndividual(index)` vrátí referenci na jedince na pozici *index*
- `SetIndividual(index, individual)` přepíše jedince na pozici *index* jedincem *individual*



Obrázek 4.3: Rozhraní IPopulation

4.2.4 ICostFunction - cenové (fitness) funkce

Cenová funkce je navržena jako funkční objekt, tj. pokud bude potřeba volat cenovou funkci, vezme se instance objektu, který toto rozhraní implementuje, a zavolá se jako funkce. Toto odpovídá syntaktickému i sémantickému hledisku, protože cenová funkce je *funkce*, a tudíž by se měla i v rámci kódu volat jako *funkce*, ale zároveň je požadováno, aby byla objektem, čehož lze dosáhnout jedině pomocí deklarace třídy. Vlastnost jazyka C++ přetěžovat operátory je zde použita tak, aby vnikl tzv. „funkční objekt“, což je v podstatě třída definující zároveň operátor volání funkce `()`.

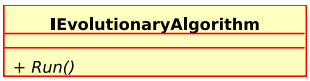


Obrázek 4.4: Rozhraní ICostFunction

- `operator()(individual,specimen)` ohodnotí jedince *individual* reálným nezáporným číslem, *specimen* slouží ke zjištění mezí a typů jednotlivých míst na chromozómu jedince.

4.2.5 IEvolutionaryAlgorithm

Jedná se o jednoduché rozhraní pro jakýkoliv evoluční algoritmus. Jedinou metodou je metoda `Run()`, která spustí samotný výpočet. Všechna ostatní nastavení parametrů algoritmu jsou záležitostí třídy, která toto rozhraní implementuje.



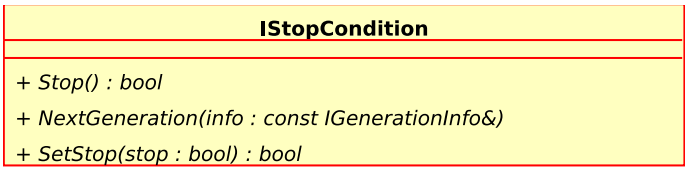
Obrázek 4.5: Rozhraní IEvolutionaryAlgorithm

- `Run()` spustí evoluční algoritmus

4.2.6 IStopCondition

Rozhraní pro podporu uživatelsky definovatelné podmínky ukončení evolučního algoritmu. Uživatel zdědí od tohoto rozhraní vlastní třídu, která na základě informací jí předávaných rozhodne, jestli už nastala vhodná doba k ukončení výpočtu. Ať již bude požadováno

ukončení na základě dosažené ceny, generace, počtu volání cenové funkce nebo třeba i složitějších podmínek(žádné zlepšení za posledních 20 generací, ...), bude možnost toto specifikovat ponechána na samotnou konkrétní aplikaci. Toto řešení představuje jiný přístup, oproti možná klasickému řešení pomocí předem vytvořené sady nastavitelných podmínek. Takto koncipovaná podmínka bude předána (resp. reference na její rozhraní) evolučnímu algoritmu, který ji bude využívat.

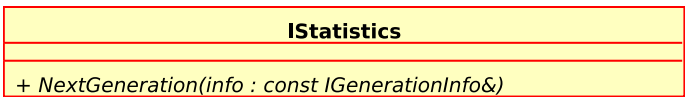


Obrázek 4.6: Rozhraní IStopCondition

- `Stop()` vrací true, pokud má evoluční algoritmus skončit, jinak false
- `NextGeneration(info)` voláno ev. algoritmem, předá objektu informace o nové generaci
- `SetStop(bool)` nastaví, jestli mají následující volání Stop() vrátit true

4.2.7 IStatistics - statistika

Objekt třídy, jenž bude implementovat toto rozhraní, bude sloužit ke sbírání statistik z průběhu evolučního algoritmu.



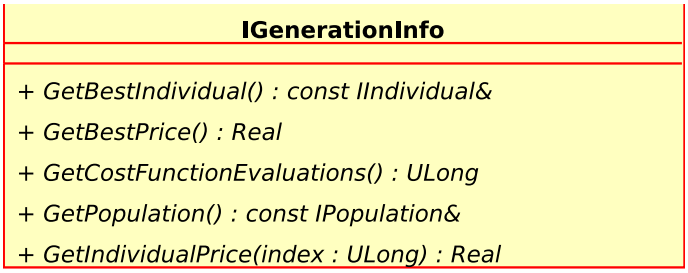
Obrázek 4.7: Rozhraní IStatistics

- `NextGeneration(info)` předá objektu informace o proběhnuvší generaci evoluční algoritmu

4.2.8 IGenerationInfo - informace o změnách během jedné generace

Rozhraní slouží jako základ pro objekt předávající informaci o změnách, k nimž došlo v rámci jedné generace běhu evolučního algoritmu.

- `GetBestIndividual()` vrátí referenci na rozhraní nejlepšího jedince
- `GetBestPrice()` vrátí cenu nejlepšího jedince
- `GetCostFunctionEvaluations()` vrátí počet volání cenové funkce
- `GetPopulation()` vrátí referenci na aktuální populaci (může být i pomocná v rámci evolučního algoritmu)



Obrázek 4.8: Rozhraní IGenerationInfo

- `GetIndividualPrice(index)` vrátí cenu jedince na pozici index v populaci získané pomocí `GetPopulation()`

Kapitola 5

Popis implementace

V této kapitole bude popsána implementace vybraných evolučních algoritmů a problémy, které poslouží k demonstraci zrychlení výpočtů v důsledku paralelizace.

5.1 Třídní hierarchie

Z rozhraní, jenž bylo popsáno v předchozí kapitole, byly poděděny třídy, jenž ho implementují. Pro každé rozhraní existuje třída, která z něho dědí. Jednotlivé třídy jsou rozebrány níže.

5.1.1 CSpecimen

Dědí ze třídy ISpecimen, a implementuje její rozhraní. Zajímavou vlastností pro podporu paralelismu je, že pokud je výpočet rozložen mezi více stanic(pomocí OpenMP), je první dimenze prohledávaného prostoru rozdělena počtem stanic tak, aby na každou vyšla menší část, tj. máme-li N stanic a hledáme extrém funkce $f(x_1, x_2, \cdots)$ na intervalu $x_1 \in \langle l_1, h_1 \rangle$, $x_2 \in \langle l_2, h_2 \rangle, \cdots$, pak je první dimenze rozdělena tak, že na i-té stanici je začínající populace vygenerována, v rozmezí hodnot $\left\langle l_1 + i\frac{h_1 - l_1}{N}, l_1 + (i+1)\frac{h_1 - l_1}{N} \right\rangle$. Toho je docíleno správnou implementací funkce `GetRandom(dimense)`, na kterou musí být pří generování jedinců populace spoléháno. Při samotném běhu algoritmu není jedincům nijak bráněno v tom, aby se pohybovali v rámci původního intervalu $\langle l_1, h_1 \rangle$, pouze jejich počáteční umístění částečně determinuje, kam se bude ubírat jejich vývoj.

Experimentálně bylo zjištěno, že daný segment prohledávaného prostoru je více zahuštěn, a tudíž i důkladněji prohledán. Další vlastností takovéhoto rozdělení je, že na jedné stanici, jejíž segment první dimenze obsahuje globální extrém, je zvýšená pravděpodobnost nalezení globálního extrému v dřívějším čase oproti stanicím ostatním. Tento závěr je odvozen na základě experimentování, a nemusí být obecně platný. Pokud je prohledávaných dimenzí vysoký počet, nelze o vhodnosti či nevhodnosti takovéto úpravy říct cokoliv konkrétního. Rozhodně je ale vhodné, aby interval dělené dimenze byl největší ze všech intervalů omezujících prohledávaný prostor, toto ale nebylo implementováno.

Zajímavou možností by bylo místo takovéhoto dělení prohledávaného prostoru využít některého nerovnoměrného rozdělení (např. podle Gaussovy křivky). To by vedlo k vytvoření populací, jejíž jedinci se po vytvoření vyskytují převážně v preferované oblasti, a zároveň by nebyla znížena pravděpodobnost vzniku „evoluční novinky“, protože by občas docházelo k páření s jedinci vzdálenými od centra populace, kteří by se v ní také s určitou

četností vyskytovali.

V rámci třídy CSpecimen tedy dochází k pokusu rozdělit problém na několik menších, a tyto pak řešit samostatně. Bylo vypozorováno, že efektivita výše nastíněného postupu záleží jak na řešeném problému, tak na použitém evolučním algoritmu. Role náhody je bohužel také velice významná.

5.1.2 CIndividual

V rámci problematiky optimalizace je jedinec v podstatě vektor reálných čísel. Nepřekvapí proto, že jako vnitřní datová reprezentace jedince byla zvolena datová struktura *vector* z knihovny STL(standard template library). Tato struktura má prakticky všechny vlastnosti klasického pole, a navíc i metody, které zjednodušují práci s ní. Jako datový typ je použit typ *Real*, což je v podstatě přejmenovaný základní typ *double*. Díky všudypřítomnosti typu *Real* je možno z jednoho místa zdrojového kódu měnit, s jakým reálným typem algoritmy pracují.

5.1.3 CPopulation

Tato třída implementuje rozhraní IPopulation, z něhož dědí. Zapouzdřuje vektor jedinců, a obsahuje také referenci na specimen, podle kterého je v konstruktoru vygenerována počáteční generace.

5.1.4 CCostFunction - cenové (fitness) funkce

Protože cenová funkce v podstatě implementuje čistě virtuální třídu ICostFunction, a tato implementace je závislá na problému, který hodláme řešit, jediné, co přidává třída *CCost-Function* je virtuální destruktor, který sám o sobě nesměl být součástí ICostFunction, protože by tím tato třída přestala být čistě abstraktní, což by porušilo cíl udělat rozhraní co nejvíce abstraktní, a přiblížit se tak modernějším objektově orientovaným jazykům, které na rozdíl od $C++$ podporují rozhraní nativně. Důvodů, proč by měl být destruktor virtuální, je více - jednak je třeba zajistit, že v případě uživatelské alokace paměti v konstruktoru potomka proběhne rovněž dealokace korektně (=zavolá se správný destruktor), a pak se také prostě doporučuje veřejný destruktor dělat virtuální. Od CCostFunction budou dědit funkce, které představují ukázkové problémy.

5.1.5 CStopCondition

Třída dědí z IStopCondition, a její jediný účel je implementace logiky metody *SetStop(bool)* a výchozí definice rozhraní tohoto rozhraní.

5.1.6 CGenerationInfo - informace o změnách během jedné generace

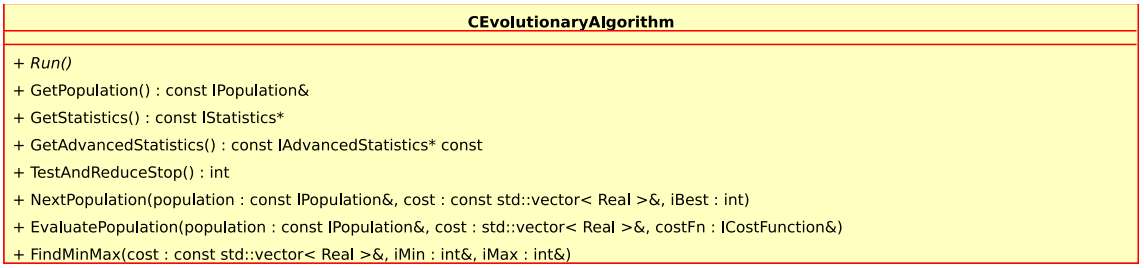
Jedná se v podstatě o objekt uchovávající data a reference na jiné objekty. Takto uchovávané informace poskytuje pomocí rozhraní IGenerationInfo, z něhož třída CGenerationInfo dědí.

5.1.7 CStatistics - statistika

Třída pro sbírání jednoduchých statistických údajů, v tomto případě pouze nejlepšího jedince, jeho ceny, počet generací a počet volání cenové funkce.

5.1.8 CEvolutionaryAlgorithm

Třída představuje obecný, zatím nespecifikovaný evoluční algoritmus. Obsahuje všechny členské proměnné, které mohou být v potomcích vyžadovány. Rovněž jsou přítomny některé pomocné funkce, jako je třeba výpočet ceny celé populace. Tím se samozřejmě ulehčí samotné programování odvozené třídy implementující evoluční algoritmus, protože mnohé z funkcionality je již implementováno v této třídě. Samotná třída dědí z rozhraní IEvolutionaryAlgorithm, ale jeho implementaci přenechává svým potomkům.



Obrázek 5.1: Rozhraní IEvolutionaryAlgorithm

Mezi významné metody této třídy patří ty, které pomáhají implementovat podporu paralelismu. Metody předpokládají, že budou volány z bloku, který je již paralelizován pomocí *#pragma omp parallel*, což je efektivnější přístup k paralelizaci pomocí OpenMP, než mít pro každou smyčku vlastní *#pragma omp parallel for*.

metoda EvaluatePopulation(population, cost, costFn)

Tato metoda slouží k výpočtu ceny populace. Výpočet probíhá paralelně pomocí rozhraní OpenMP, před smyčku for je přidána odpovídající pragma.

```
void PGALib::CEvolutionaryAlgorithm::EvaluatePopulation(
  const IPopulation& population, std::vector<Real>& cost, ICostFunction& costFn)
{
  const int size = population.GetSize();
  int i;

  #pragma omp for private(i)
  for (i=0; i<size; i++)
    cost[i] = costFn(population.GetIndividual(i), population.GetSpecimen());
  #pragma omp single
  m_CostFunctionEvaluations += size;
}
```

Výpočet cen jednotlivých jedinců populace je paralelizován způsobem, kdy cenová funkce pro různé jedince je volána paralelně více vlákny. Tento způsob paralelizace je vhodný zejména pro takové cenové funkce, jejichž výpočet trvá dlouho, a režie pro paralelizaci nepřevýší časovou ztrátou zisk daný paralelizací výpočtu.

metoda FindMinMax(cost,index_min,index_max)

Metoda paralelně nalezne indexy minima a maxima v poli *cost*. Hledání je provedeno tak, že do lokálních proměnných metody se uloží indexy nalezené v části pole dané paralelizací, potom první vlákno, které dojde do části označené pragmou single uloží do výstupních indexů svoje nalezené, a následuje kritická sekce, kdy každé vlákno porovná hodnotu v poli

s výstupním indexem s hodnotou v poli danou svým lokálním indexem, a když je jeho řešení lepší, aktualizuje výstupní index.

```
void PGALib::CEvolutionaryAlgorithm::FindMinMax(
  const std::vector<Real>& cost, int& iMin, int& iMax)
{
  const int size = cost.size();
  int i, indexMin = 0, int indexMax = 0;
  Real minCost = RealMax; Real maxCost = RealMin;

  // nalezení minimální ceny pro dané vlákno
  #pragma omp for private(i)
  for (i=0; i<size; i++) {
    if (minCost > cost[i]) { minCost = cost[i]; indexMin = i; }
    if (maxCost < cost[i]) { maxCost = cost[i]; indexMax = i; }
  }

  // první vlákno nastaví výstup metody jako své nalezené extrémy
  #pragma omp single
  { iMin = indexMin; iMax = indexMax; }

  // pak každé vlákno porovná své nalezené extrémy s~výstupem
  // a případně upraví výstup
  #pragma omp critical(FINDMINMAX)
  {
    if (cost[iMin] > minCost) iMin = indexMin;
    if (cost[iMax] < maxCost) iMax = indexMax;
  }
}
```

NextPopulation(population, cost, index_best)

Metoda, která se volá při přestupu z jedné generace do druhé. Aktualizuje vnitřní čítač generací *m_Generation*, čítač počtu volání cenové funkce *m_CostFunctionEvaluations* a vytvoří a předá objekt statistikám, čímž je informuje o aktuálně proběhnuvším kroku evolučního algoritmu.

TestAndReduceStop()

Tato metoda slouží k synchronizaci podmínky ukončení algoritmu mezi více MPI procesy. Každý proces má svojí ukončovací podmínku *m_StopCondition*, která může vracet true a nebo false. Pokud alespoň v jednom procesu nastane situace, že ukončovací podmínka je pravdivá, pak je výpočet třeba ukončit na všech procesech, protože cíle již bylo dosaženo. Toho se docílí pomocí funkce *MPI_Allreduce* s operátor logické nebo. K samotné synchronizaci dochází pouze tehdy, je-li splněna podmínka ukončení, nebo jednou za určitý počet generací. Pokud tedy v jednom procesu dojde ke splnění podmínek ukončení, pak tento se zastaví na volání *MPI_Allreduce*, kde čeká, dokud v ostatních procesech není tato funkce rovněž zavolána. Metoda pak nastaví objektu *m_StopCondition*, zdali má při příštím volání své metody *Stop()* vrátit true.

5.1.9 MigrationBest

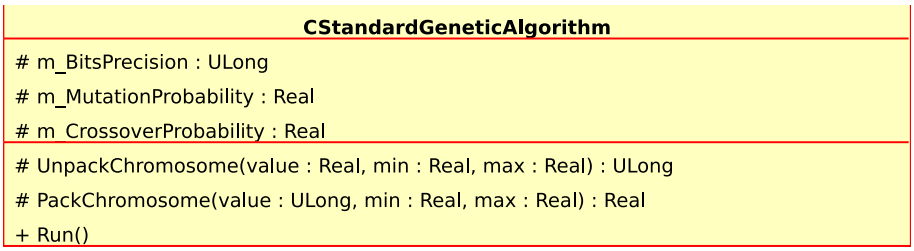
V této metodě si populace pomocí MPI vymění nejlepší jedince, kteří nahradí prvního jedince, který má horší ohodnocení než oni. Pomocí funkce *MPI_Allgather* jsou tito jedinci rozhlášeni mezi všechny procesy. Spolu s jedinci je zasílána i jejich cena, aby ji nebylo nutno znova počítat v cílových procesech.

5.1.10 MigrationRound

Metoda *MigrationRound* veme část populace, a přepošle ji procesu s indexem o jedna vyšší pomocí neblokující funkce *MPI_Isend*. Poté čeká na přijetí části generace od procesu s indexem o jedna nižším(nebo posledním) pomocí funkce *MPI_Recv*. Část populace, jenž byla poslána pryč, je nahrazena příchozí populací.

5.2 CStandardGeneticAlgorithm - Standardní genetický algoritmus

Třída *CStandardGeneticAlgorithm* dědí z *CEvolutionaryAlgorithm* a implementuje metodu *Run()* a další pomocné metody, které potřebuje pro svou funkci.



Obrázek 5.2: Rozhraní IEvolutionaryAlgorithm

Celá evoluce se odehrává ve while cyklu, podmínka jeho ukončení je dána jako volání metody stop objektu implementujícího rozhraní IStopCondition, referenfe na nějž byla předána jako parametr konstruktoru objektu, a která se uchovává v *m_StopConditon*. Samotný cyklus je paralelizován pomocí OpenMP direktivou *#pragma omp parallel*.

5.2.1 Popis algoritmu

1. Výběr jedinců pro crossing-over: Pro každého jedince v populaci se vygeneruje náhodné číslo, a pokud je toto menší než pravděpodobnost křížení, je jedinec zařazen mezi jedince určené ke křížení. Pokud je počet takto vybraných jedinců nakonec lichý, je poslední jedinec odebrán. Tato akce se provádí pouze v jednom vláknu, a vybraní jedinci jsou přidání do sdíleného pole.

2. Crossing-over - jednobodové křížení: Pro každou dvojici po sobě jdoucích jedinců vybraných v předchozím kroku algoritmu se provede tzv. jednobodové křížení, kdy se vybere náhodný index který oba jedince rozdělí na dvě různé části nenulové délky, a pravé dvě strany (tj. od indexu do konce jedince) si prohodí hodnoty chomozómů.

3. Ohodnocení populace: Populace je ohodnocena pomocí metody předka *EvaluatePopulation*, ceny jsou uloženy do vektoru *m_Cost*.

4. Nalezení extrémních jedinců: Pomocí metody předka *FindMinMax* jsou nalezeny indexy nejlepšího a nejhoršího nalezeného jedince. Následně jsou nejlepší a nejhorší jedinec jedním z vláken zkopírováni do členských proměnných.

5. Aktualizace statistik a krok k další populaci: Voláním metody předka *NextPopulation* je provedena aktualizace statistik. Dá se říct, že zde končí jedna generace algoritmu,

což je možná poněkud netradiční, ale bylo to nutné, aby se nemusela ohodnocovat populace před samotným započetím paralelního *while* cyklu, čímž se předejde duplikaci kódu v programu, navíc by nebylo dvakrát efektivní vytvářet paralelní sekci, v ní ohodnotit jedince, tu pak zrušit, a následně hned vytvářet další pro cyklus *while*.

6. Redukce ukončovací podmínky pomocí MPI: Zavoláním metody *TestAndReduceStop()* předka dojde k sjednocení stavu ukončovací podmínky algoritmu. Pokud má algoritmus skončit, následující kroky algoritmu se nevykonají. Tato část algoritmu je provedena pouze hlavním vláknem.

7. Generování rulety: Ruleta je pole určité vhodné velikosti (v této implementaci 32768), které obsahuje indexy jedinců v populaci. Je vyžadováno, aby se index kvalitnějších jedinců v tomto poli vyskytoval častěji, než index jedinců méně kvalitních - tj. aby incidence indexu nějakého jedince v poli byla přímo úměrná jeho kvalitě. Pro cenovou funkci, jenž je využívaná v rámci implementace, však platí, že čím menší je hodnota, kterou vrátí, tím kvalitnější je daný jedinec. Vzorec pro výpočet pravděpodobnosti daného jedince v ruletovém kole však předpokládá pravý opak - čím vyšší hodnota, tím vyšší kvalita jedince. Je proto třeba nejdříve přepočítat pole cen tak, aby nejnižší hodnota byla nejvyšší a naopak. Toho se dosáhne tak, že se hodnota v poli cen přepíše rozdílem nejhorší (=nejvyšší) ceny s cenou na daném místě v poli ($cost[i] = worstCost - cost[i]$). Předpokládá se, že cenová funkce nikdy nevrací zápornou hodnotu.

   Po provedení výše nastíněné úpravy cen se pro každého jedince vypočítá jeho pravděpodobnost postoupení do další generace, podle vzorce $P(i) = \frac{cost[i]}{\sum_{j=0}^{n} cost[j]}$. Následně se podle pravděpodobností naplní pole rulety odpovídajícími indexy. Jedinci s indexem i je přiřazeno $P(i) * velikost_pole_ruletty$ políček rulety, tj. zapíše se na ně jeho index.

8. Výběr nové populace podle rulety a provedení mutace: Pro každou pozici v nové generaci je vygenerováno náhodné číslo indexující pole rulety, podle něhož je vybrán jedinec, který bude do nové generace zkopírován. Během kopírování je každé místo chromozómu jedince převedeno do vhodné binární reprezentace, a nad každým bitem je proveden náhodný pokus pravděpodobnosti mutace takový, že pokud uspěje, dojde k inverzi daného bitu. Takto upravené místo chromozómu je potom převedeno zpět do reálné reprezentace, a uloženo na odpovídající pozici v novém jedinci. Najednou tak probíhá jak výběr nové populace, tak mutování jejich jedinců. Místo chromosomu s indexem i je převedeno na binární reprezentaci podle vzorce

$$unpacked = \frac{chromosome_i - min_i}{max_i - min_i} 2^{32}$$

   a zabaleno zpět podle

$$packed = min_i + (max_i - min_i) \frac{unpacked}{2^{32}}$$

   .

   Po výběru nové populace je někam náhodně umístěn nejlepší jedinec z generace předchozí. Tím je zajištěno, že nejlepší dosud nalezené řešení nebude náhodou ztraceno.

9. Výměna jedinců pomocí MPI Na tomto místě bylo zkoušeno několik metod, které umožňují výměnu jedinců pomocí MPI. Nakonec zde byla umístěna metoda *MigrateRound()*.

5.3 Diferenciální evoluce

Implementace diferenciální evoluce je vůbec první z algoritmů, jenž byly v rámci této práce implementovány, a proto je méně čistá - všechen kód je obsažen v metodě *Run()*. Kusy tohoto kódu později posloužily k vytvoření metod v třídě *CEvolutionaryAlgorithm*, které posléze byly použity k implementace dalších evolučních algoritmů. Protože však samotný kód diferenciální evoluce fungoval dobře, nebyly v něm již další změny prováděny.

Samotná implementace diferenciální evoluce je provedena ve třídě *CDifferentialEvolution*. poděděné ze třídy *CEvolutionaryAlgorithm*. Diferenciální evoluce má dva parametry - parametr F a CR, z nichž první je váhový faktor, a druhý je křížící konstanta. Tyto jsou nastavitelné v konstruktoru, spolu s parametry, které jsou nutné k inicializaci konstruktoru předka.

Algoritmus pro svou práci potřebuje populace dvě, z nichž jedna je pomocná, a proto je vytvořena pomocná populace, která je kopií generace předané v konstruktoru. Na tyto dvě generace poté ukazují dva ukazatele, z nichž jeden vždy plní funkci generace aktuální, a druhý generace vytvářené. Vždy po jedné generaci jsou ukazatele prohozeny, aby se nemusela kopírovat celá generace výstupní do generace vstupní. Po skončení algoritmu je otestováno, jestli ukazatel na výstupní generaci ukazuje na generaci z konstruktoru, a pokud ne, je do ní pomocná generace nakopírována.

1. Počáteční ohodnocení populace: Před započetím samotného cyklu algoritmu je do pole *cost* ohodnocen každý jedinec populace. Velikost pole je tedy rovna velikosti populace, s níž pracujeme. Smyčka *for*s výpočtem cenové funkce je paralelizována pomocí OpenMP.

2. Cyklus algoritmu: Cyklus algoritmu probíhá ve smyčce *while*, dokud metoda *Stop()* objektu *m_StopCondition* nevrátí hodnotu true. Cyklus je paralelizován pomocí OpenMP, jsou uvedeny seznamy proměnných soukromých i sdílených. Všechen kód uvnitř cyklu, není-li uvedeno jinak, probíhá paralelně ve více vláknech.

3. Páření: Vytváření nových jedinců probíhá tak, že pro každý index v cílové populaci se vyberou čtyři jedinci, z nichž jeden je v původní generaci na stejném indexu, a další tři mají jiné vzájemně různé náhodné indexy. Potom se podle vzorce uvedeného v dřívější kapitole vygeneruje zkušební jedinec, a pokud je jeho ohodnocení lepší než jedinec původní, je zařazen do nové generace, jinak do nové generace postupuje původní jedinec.

4. Nalezení nejlepšího: Nejlepší jedinec je nalezen jednoduchým cyklem provedeným v hlavním vláknu.

5. Předání informací statistikám a krok k další generaci: Statistiky jsou aktualizovány pomocí objektu *CGenerationInfo*, a je proveden přechod k další generaci. Ukazatele na zdrojovou a cílovou populaci jsou prohozeny. Rovněž je aktualizována podmínka pro ukončení algoritmu na rovněž základě *CGenerationInfo*.

6. Výměna jedinců pomocí MPI: pomocí *MigrateRound()*.

7. Redukce ukončovací podmínky algoritmu mezi procesy MPI:pomocí metody předka *TestAndReduce()*.

5.4 Samoorganizační migrační algoritmus

Algoritmus SOMA je implementován ve třídě *CSoma*, která dědí ze třídy *CEvolutionaryAlgorithm*. V konstruktoru objektu je volán jednak konstruktor předka z požadovanými parametry(ty jsou takto vynuceny i v konstruktoru potomka), jednak jsou nastaveny členské proměnné specifické pro daný problém - jsou jimi *m_Mass*, *m_Step* a *m_Prt*, mající význam popsaný v teoretické části této práce. Hlavní část algoritmu je jako obvykle zapsána v metodě *Run()*, ale význam mají také metody *GenPRTVector* a *AllToOne*.

1. Inicializace algoritmu: Nejdříve je celá populace ohodnocena, tak aby byly známy ceny jednotlivců. Poté jsou nalezeni nejlepší a nejhorší jedinec. To vše je vykonáno pomocí metod předka *EvaluatePopulation* a *FindMinMax*.

2. Hlavní cyklus: algoritmus je vykonáván v cyklech, dokud nedojde ke splnění ukončovací podmínky *m_StopCondition.Stop()*.

3. Pro každého jedince kromě jedince hlavního je zavolána funkce *AllToOne*. Volání této stěžejní funkce je prováděno paralelně pro každého jedince zvlášť pomocí OpenMP. Toto je učiněno proto, že pro každého jedince bude mnohokrát počítána jeho cenová funkce, což je situace, kdy je velmi vhodné paralelizovat. Účelem funkce *AllToOne* je hledat nové umístění jedince na hyperploše prohledávané oblasti, a to takové, jehož ohodnocení je lepší. Metodou *GenPRTVector* je vytvořen perturbační vektor, který sníží počet dimenzí prohledávaného prostoru. Následuje cyklus, který sleduje dráhu ve směru od aktuálního jedince k nejlepšímu, ale jenom v těch rozměrech, kde má perturbační vektor hodnotu 1. Dráha může následovat i „za“ nejlepšího jedince, nejsrozumitelněji asi zní, že při prohledávání „přestřelíme“ za nejlepšího jedince. Sledování dráhy funguje tak, že se pohybujeme směrem k nejlepšímu jedinci s nějakým malým krokem *m_Step*. Během této cesty je pravidelně ohodnocován, a nalezen nejlepší nový jedinec. Tímto je pak nahrazen jedinec původní.

4. Pomocí metody *FindMinMax* je nalezen index nejlepšího a nejhoršího jedince. Následně jsou jedním z vláken aktualizovány objekty uchovávající nejlepšího a nejhoršího jedince a jejich ceny.

5. Následuje krok k další generaci, jsou aktualizovány statistiky a objekt *m_StopCondition*, který rozhoduje o ukončení algoritmu na základě svých vlastních kritériích, odvozených od průběžně dodávaných statistických údajů.

6. Dojde k redukci ukončovacích podmínek MPI procesů pomocí metody *TestAndReduceStop()* předka.

7. Úplně na konci algoritmu je volána metoda *MPI_Barrier(MPI_COMM_WORLD)*, která zaručí, že všechny procesy opustí volání metody *Run()* ve stejnou chvíli.

5.5 Stochastický horolezecký algoritmus

Je implemetován tak, aby fungoval podle způsobu popsaného v sekci popisující algoritmy.

5.6 Problémy pro řešení pomocí paralelních evolučních technik

CCostFractal - inverzní fraktální problém

Prvním problémem, na němž bude předvedena paralelizace evolučních algoritmů, je inverzní fraktální problém(IFP). IFP je úloha, kdy máme bitmapu s vyobrazením nějakého druhu fraktálu, který byl vygenerován s nějakými koeficienty. Cílem je nalézt tyto koeficienty. Nechť koeficienty jsou dány jako uspořádaná n-tice reálných čísel, představující jedince. Pak je-li podle něj vygenerována bitmapa, zobrazující fraktál stejného typu, jako je ten, pro nějž hledáme koeficienty, je cena jedince dána jako suma absolutních hodnot rozdílů mezi sobě si odpovídajícími body bitmapy vygenerované a bitmapy s hledaným fraktálem. Protože generování fraktálu je problém obecně náročný, je třeba jednak minimalizovat nutný počet ohodnocení zkušebních jedinců, což znamená, že je nutno zvolit správný algoritmus. Dalšího urychlení je možno dosáhnout, pokud se provede paralelizace. IFP se dobře paralelizuje hlavně pomocí OpenMP, protože výpočet cenové funkce je výrazně náročnější, než režie spojená paralelizací smyčky (tohoto závěru bylo dosaženo „a posteriori“ na základě experimentu). Podrobnější výsledky budou prezentovány později.

Třída CCostFractal tvoří základ pro dědění ostatních tříd, jenž upřesní výpočet potřebného fraktálu. Hlavní její vlastností je, že v konstruktoru se jí předá bitmapa hledaného fraktálu. Obsahuje rovněž metody pro uložení vygenerovaného obrázku na disk.

CCostMandelbrot - Mandelbrotova množina

Tato třída dědí z CCostFractal, a je schopna generovat mandelbrotovu množinu podle jedince, kterého má ohodnotit. Rozdíl mezi hledaným fraktálem a takto vygenerovaným je předán jako výsledek volání objektu tohoto typu jako funkčního objektu.

Mandelbrotova množina je množina bodů v komplexním prostoru, což je v podstatě dvourozměrný eukleidovský prostor doplněný o aritmetická pravidla pro práci s komplexními čísly(zjednodušeně řečeno). Chceme-li o nějakém bodu $x = \langle x_{real}, x_{imag} \rangle$ rozhodnout, zdali patří do Mandelbrotovy množiny, vezmeme dvě komplexní čísla z_0, c takové, že $z_0 = c = x$. Potom provádíme iterace takové, že $z_{n+1} = az_n^b + c$, a pokud i po mnoha (např. 256) iteracích platí $abs(z_n) < 2.0$, prohlásíme daný bod za bod Mandelbrotovy množiny. Pokud dojde k úniku bodu mimo takto definovanou oblast, bod do množiny nepatří, ale ve výsledné bitmapě ho obarvíme barvou odpovídající počtu iterací. Je-li barevné spektrum spojité, vzniknou překrásné obrazce s barevnými přechody. Výše uvedený popis zajisté není z matematického hlediska příliš korektní, obsahuje však všechno nutné, co potřebuje kodér vědět o Mandelbrotově množině.

Jako úloha byla vygenerována bitmapa fraktálu s koeficienty $a = 1.98765$ a $b = 2.12345$. Úkolem evolučního algoritmu bude nalézt tyto koeficienty, aniž by je předem znal. Algoritmu bude k dispozici pouhý přiblížený výřez celého fraktálu, tj. nebude znám celkový vzhled fraktálu, ale jen jeho určitá oblast (x od -1 do -0.5, y od -0.5 do 0).

CCostFourier - hledání koeficientů Fourierovy řady

Dalším prezentovaným problémem bude hledání koeficientů Fourierovy řady. Lze dokázat, že každá periodická funkce může být aproximována pomocí řady čísel

$$f(x) \approx a_0/2 + a_1 \cos(x) + b_1 \sin(x) + a_2 \cos(2x) + b_2 \sin(2x) + ...$$

. Čím více je členů takovéto řady, tím více je aproximace přesnější. V některých případech je řada dokonce konečná. Hledání koeficientů je typický multidimenzionální problém. Čím přesnější aproximaci chceme, tím větší prostor bude nutno prohledat. Problém se tedy zdá být dostatečně výpočetně náročný, aby posloužil k účelu demonstrace paralelizace. Na druhou stranu je pravda, že existují lepší metody hledání koeficientů Fourierovy řady. Tento příklad má tudíž spíše demonstrační charakter, a nepředstavuje vhodný způsob řešení tohoto problému. Funkce, jejíž koeficienty hledáme, byla zvolena ve tvaru

$$f(x) = 3.1+$$

$$2.5\cos x + 3.2\cos 2x + 1.6\cos 3x + 2.8\cos 4x - 4.2\cos 5x + 5.5\cos 6x$$

$$-1.2\sin x + 3.0\sin 2x + 2.1\sin 3x - 2.98\sin 4x + 3.1\sin 5x + 1.95\sin 6x$$

a je tedy patrné, že již je ve tvaru konečné Fourierovy řady, a to proto, aby bylo snadné zkontrolovat výsledek. Jako meze pro jednotlivé koeficienty byl zvolen interval $\langle -6, 6\rangle$.

CCostBackPropagationNetwork

Jednou z poměrně složitých úloh je hledání váhových koeficientů vícevrstvé dopředné neuronové sítě. Běžně se tak činí pomocí algoritmu zpětného šíření (back-propagation), my se ale zaměříme na alternativní postup - využití evolučního algoritmu k nalezení optimálních vah a prahů. Jedním ze způsobů, jakým je zpětné šíření vysvětlováno, je minimalizace globální chyby. Ta vzniká jako suma rozdílu požadovaných výsledků od výsledků skutečných, přičemž se má zato, že čím větší je nějaký váhový koeficient, tím více přispívá k celkové chybě. Existuje zde tedy nějaké ohodnocení kvality neuronové sítě, a díky tomu se naskýtá možnost použít evoluční algoritmus. Nechť T trénovací množina, pak platí, že

$$T = \left\{ \left(\left(x_1^1, x_2^1, ..., x_n^1\right), \left(y_1^1, y_2^1, ..., y_m^1\right) \right), ..., \left(\vec{x^N}, \vec{d^N}\right) \right\}$$

Kde vektor $\vec{x^i}$ představuje vstup, a vektor $\vec{y^i}$ představuje požadovaný výstup.

Neuronová síť sestává z umělých neuronů, což jsou jednotky, které mají váhový vektor, kterým se skalárně vynásobí vstup, od výsledku se odečte tzv. práh, a to celé se dá jako vstup sigmodiální funkce, jejímž výstupem je výsledek.

$$y_i = \frac{1}{1 + e^{-\lambda \vec{x} \vec{w_i}}}$$

kde y_i je výstup neuronu i, x je vstupní vektor tvořený výstupy neuronů předchozí vrstvy, a w_i je jeho váhový vektor. Neurony jsou uspořádány do vrstev, výpočet probíhá postupně od nejnižší vrstvy k nejvyšší - nejdříve se vypočítají výstupy neuronů nižší vrstvy, které poslouží jako vstup vrstvy následující. Nejnižší vrstva sítě je vrstva vstupní, nejvyšší je výstupní, ostatní jsou skryté. Důležitou vlastností neuronové sítě je, že dovede s libovolnou přesností aproximovat libovolnou funkci. Protože problematika neuronových sítí je poměrně rozsáhlá, nebudeme se jí zde již více zabývat. Zájemce o danou problematiku nechť nahlédne do [3].

Protože neuronová síť je sama o sobě jenom nástrojem, je třeba specifikovat, čemu se bude pomocí evolučního algoritmu učit. Jako experiment bylo zvoleno sčítání dvou čtyřbitových čísel, jehož výsledkem je číslo pětibitové. Nalezení optimálního rozložení zajisté nebude jednoduché, a aby úloha byla ještě těžší, může každý neuron mít svůj vlastní parametr λ_i pro výpočet svého výstupu pomocí sigmodiální funkce.

Kapitola 6

Srovnání a grafy

V této kapitole budou popsány výsledky měření zrychlení dosaženého pomocí paralelizace. Aby bylo možno usoudit, jak která metoda paralelizace pomohla zrychlit, či zlepšit přesnost evolučních algoritmů, bude popsáno jak výsledků dosažených pouze pomocí OpenMP či MPI, tak kombinací obou metod paralelizace. Pokud nějaký výpočet paralelizujeme, máme dvě možnosti, jak s paralelizací naložit - buď díky ní snížíme dobu, která je potřebná k dosažení výsledku obdobné kvality, jako při použití sekvenční verze algoritmu, nebo necháme algoritmus běžet stejně dlouho a dosáhneme lepšího řešení ve stejném čase. První případ má význam, chceme-li snížit reakční dobu nějakého systému pracujícím v reálném čase, druhý, pokud v pravidelných intervalech dochází k hledání řešení, které je potom používáno po určitou časovou jednotku(např. plánování optimální výroby pro následující den, výpočet probíhá přes noc).

6.1 Zrychlení dosažené pomocí OpenMP na dvoujádrovém procesoru

Pro testování zrychlení výpočtů na dvoujádrovém procesoru byl použit dvoujádrový počítač mající následující parametry získané pomocí `cat /proc/cpuinfo`:

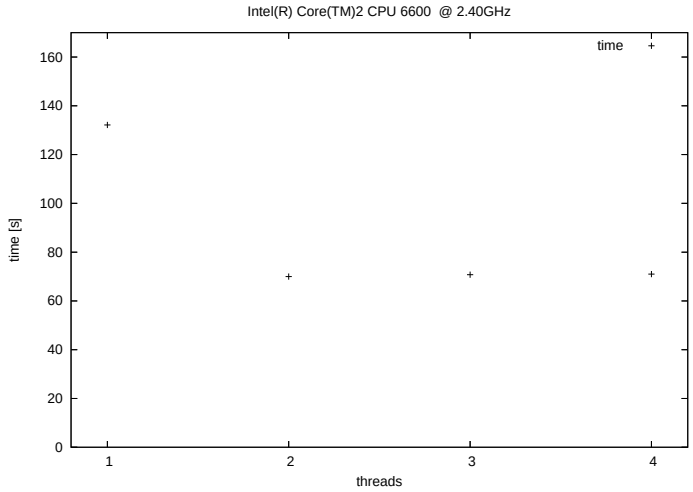
```
model name      : Intel(R) Core(TM)2 CPU          6600  @ 2.40GHz
cpu MHz         : 2394.123
cache size      : 4096 KB
cpu cores       : 2
```

Počet vláken, pro něž byly měřeny reálné časy, byl zvolen z intervalu 1 až 4. Bylo tedy měřeno i více vláken, než je počet procesorů. Ve výsledcích je obvykle vidět, že doba trvání výpočtu při více než dvou vláknech se moc neliší od doby při dvou vláknech. Pokud byla úloha příznivá způsobu paralelizace, je doba trvání paralelizované verze algoritmu zhruba poloviční, než verze sekvenční.

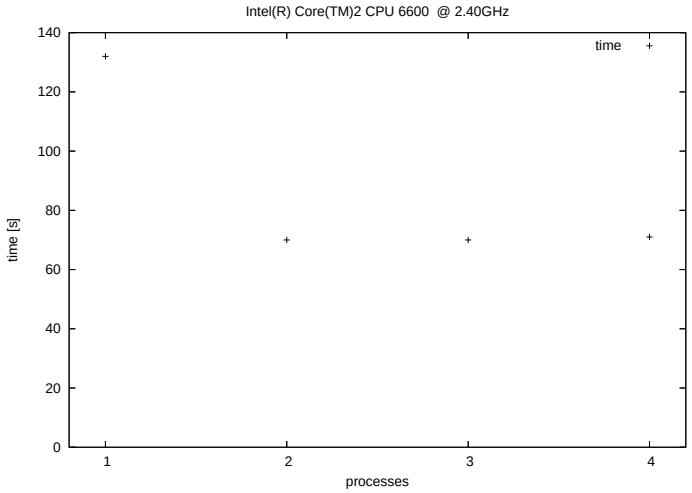
6.1.1 Inverzní fraktální problém

Inverzní fraktální problém je úloha, jenž je velice vhodná k paralelizaci. Výpočet fraktálu je značně náročný, a každému vláknu trvá poměrně dlouho. Z důvodu vyšší rychlosti výpočtu byl počet iterací pro každý bod komplexního prostoru zvolen 16. Na nalezení hledaného řešení tato volba neměla negativní vliv, a některým algoritmům se podařilo nalézt řešení

zcela přesně. Přesto doba běhů nebyla zvolena podle kvality výsledku, ale na nějakou rozumnou hodnotu o délce maximálně něco málo přes dvě minuty pro sekvenční verzi. Cílem měření bylo totiž zjistit samotné zrychlení, nikoliv kvalitu měřených algoritmů.

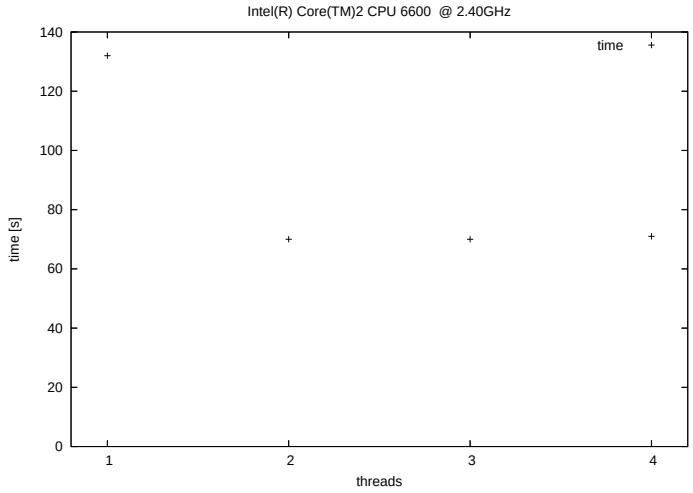


(a) SGA - hledání fraktálu (dvoujádrový procesor)

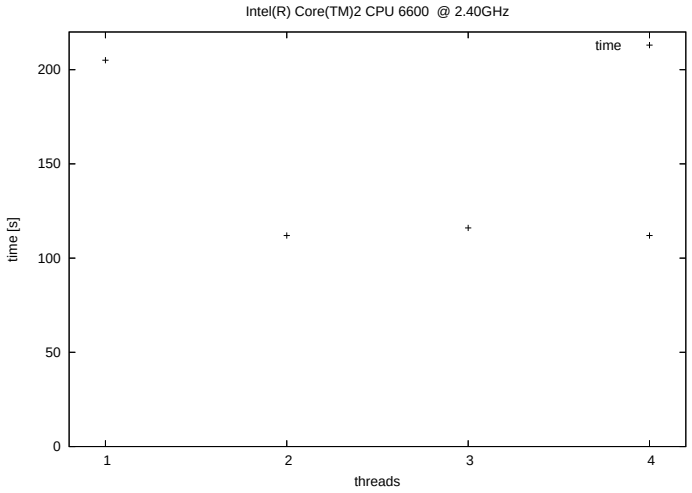


(b) DE - hledání fraktálu (dvoujádrový procesor)

Obrázek 6.1: Zrychlení výpočtu fraktálu na dvoujádrovém procesoru(SGA,DE)



(a) SOMA - hledání fraktálu (dvoujádrový procesor)



(b) SHA - hledání fraktálu (dvoujádrový procesor)

Obrázek 6.2: Zrychlení výpočtu fraktálu na dvoujádrovém procesoru(SOMA,SHA)

Standardní genetický algoritmus - SGA

Na obrázku 6.1(a) je možno sledovat zrychlení standardního genetického algoritmu na dvoujádrovém procesoru. Velikost populace byla rovna 20, pravděpodobnost mutace 5% a míra křižení 20%. Je patrné, že doba pro provedení výpočtu se z využitím obou jader zkrátila téměř na polovinu.

Diferenciální evoluce - DE

Obrázek 6.1(b) ukazuje zrychlení diferenciální evoluce. Parametry algoritmu: $F = 0.9$, $CR = 0.95$, velikost populace 20. Z obrázku je patrné, že paralelizace byla viditelným přínosem.

Samoorganizační migrační algoritmus - SOMA

Z obrázku 6.2(a) lze vyvodit, že i SOMA je dobře paralelizovatelná, pokud je účelová funkce dostatečně výpočetně náročná. Parametry algoritmu SOMA: $MASS = 2$, $STEP = 0.312$, $PRT = 0.4$.

Stochastický horolezecký algoritmus - SHA

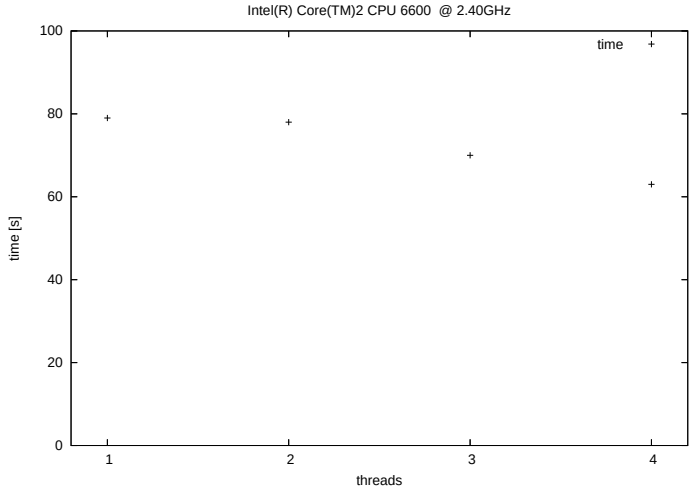
Rovněž pro SHA lze na obrázku 6.2(b) vidět, že došlo k téměř dvojnásobnému zrychlení.

Shrnutí

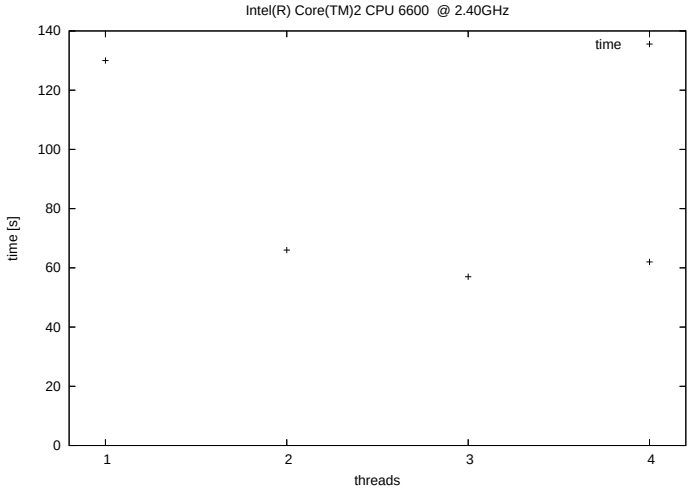
Z výše uvedených grafů je patrné, že úloha inverzního fraktálního problému byla úspěšně paralelizována.

6.1.2 Učení umělé neuronové sítě

Tato úloha se ukázala jako méně vhodná pro demonstraci, protože jednak se příliš nedaří učení sítě, jednak výsledky paralelizace nejsou tak pěkné, jak by bylo potřeba. Síť měla problém naučit se rozumně sčítat 3-bitová čísla, a tak funkční sčítání 4-bitových čísel je spíše utopií. Přesto proběhl pokus nalézt optimální váhy, prahy, a lambdy pro sít o rozměrech 4x12x5. Výsledky měření zrychlení jsou uvedeny níže.



(a) SGA - hledání vah neuronů(dvoujádrový procesor)



(b) DE - hledání vah neuronů(dvoujádrový procesor)

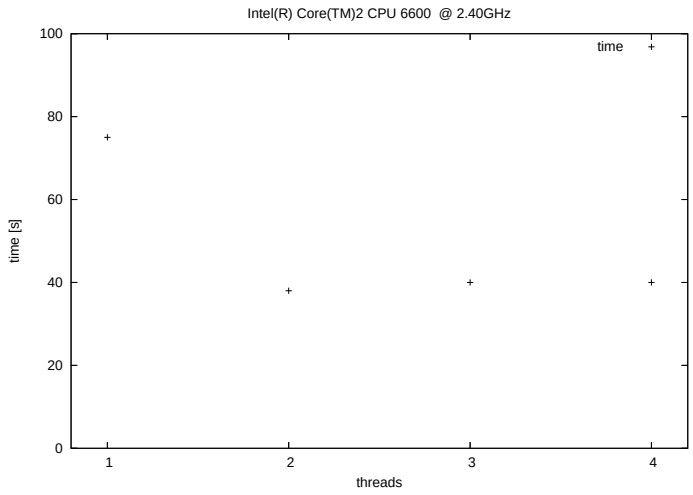
Obrázek 6.3: Zrychlení výpočtu fraktálu na dvoujádrovém procesoru(SGA,DE)

Standardní genetický algoritmus - SGA

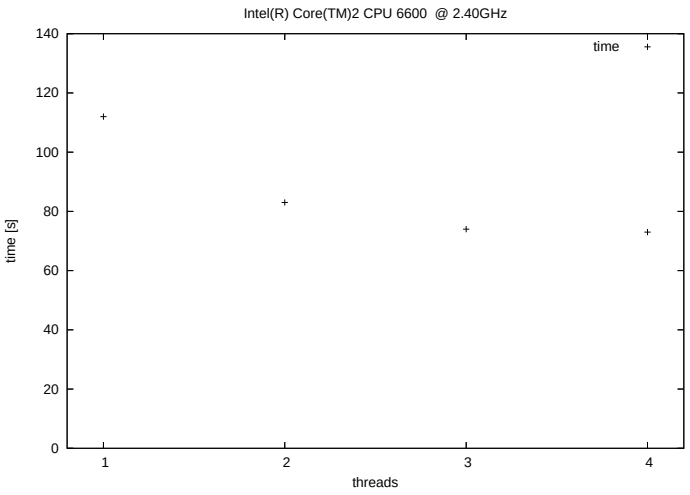
Zrychlení SGA(6.3(a)) je jen velmi málo patrné. Je to možná proto, že tato verze algoritmu příliš spoléhá na knihovní generátor náhodných čísel, který se chová, jako kdyby generování náhodného čísla probíhalo v kritické sekci. V budoucích verzích knihovny bude nutno zlepšit kvalitu implementace.

Diferenciální evoluce - DE

Diferenciální evoluce byla paralelizována úspěšně i pro tuto úlohu(6.3(b)). Doba výpočtu se zkrátila zhruba na polovinu.



(a) SOMA - hledání vah neuronů(dvoujádrový procesor)



(b) SHA - hledání vah neuronů(dvoujádrový procesor)

Obrázek 6.4: Zrychlení výpočtu fraktálu na dvoujádrovém procesoru(SOMA,SHA)

Samoorganizační migrační algoritmus - SOMA

Algoritmus SOMA byl rovněž úspěšně paralelizován, jak je vidět z grafu 6.4(a).

Stochastický horolezecký algoritmus - SHA

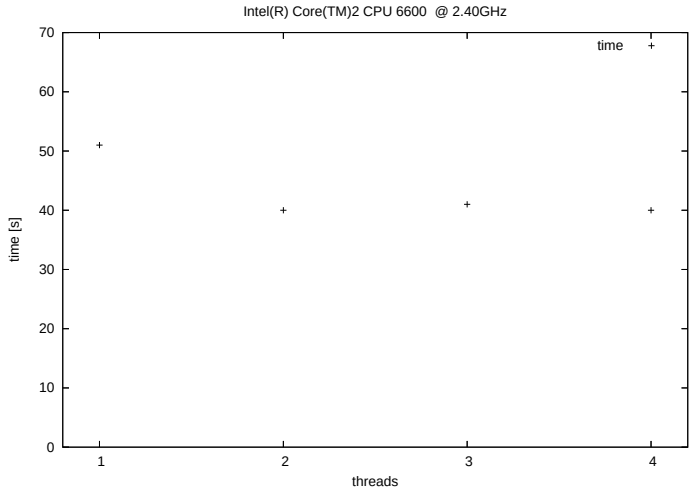
Poněkud překvapivě SHA nedopadl tak dobře, jak by se dalo očekávat - 6.4(b).

Shrnutí

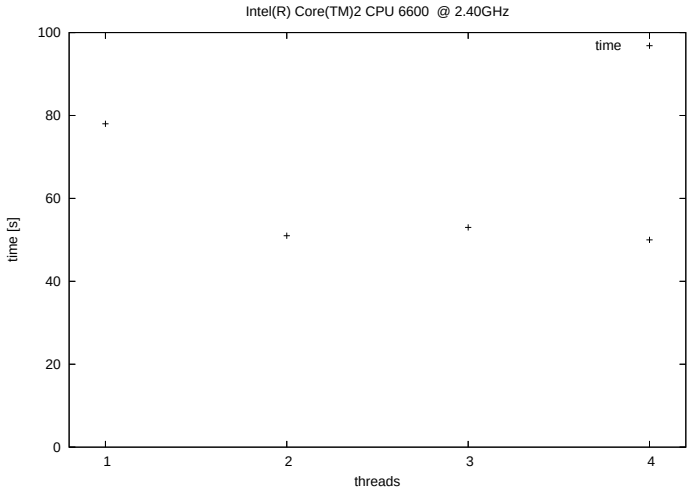
Úloha hledání vah neuronové sítě byla poněkud hůře paralelizovatelnou. S danými algoritmy tak jak jsou implementovány se zrychlení dostaví především tehdy, je-li trénovací množina dostatečně velká, nebo je velký počet vah neuronů.

6.1.3 Hledání koeficientů Fourierovy řady

Hledání koeficientů Fourierovy řady je další měřená úloha. Představuje typ úlohy, kde výpočet ceny jedince není až tak náročný, alespoň v porovnání s předešlými dvěma úlohami.



(a) SGA - hledání koeficientů Fourierovy řady (dvoujádrový procesor)



(b) DE - hledání koeficientů Fourierovy řady (dvoujádrový procesor)

Obrázek 6.5: Zrychlení výpočtu koeficientů Fourierovy řady na dvoujádrovém procesoru(SGA,DE)

Standardní genetický algoritmus - SGA

Z obrázku 6.5(a) je vidět, že zrychlení je přítomné, ale ne výrazné.

Diferenciální evoluce - DE

Obrázek 6.5(b) ukazuje adekvátní zkrácení doby výpočtu. Diferenciální evoluce tudíž sklízí úspěch.

Samoorganizační migrační algoritmus - SOMA

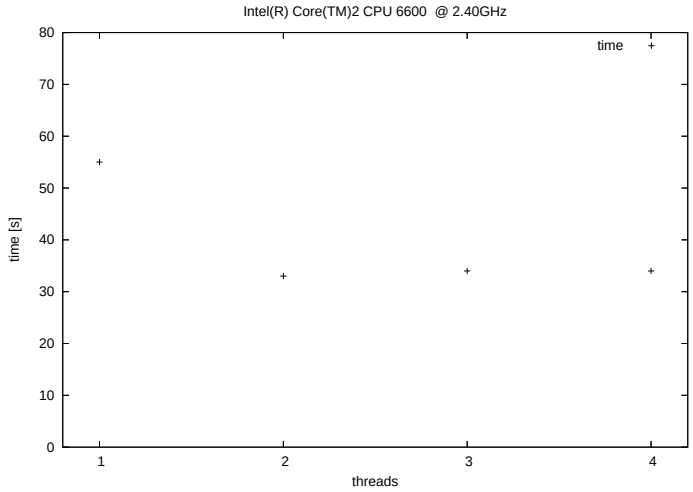
Rovněž SOMA dopadla podle očekávání dobře - viz. 6.6(a).

Stochastický horolezecký algoritmus - SHA

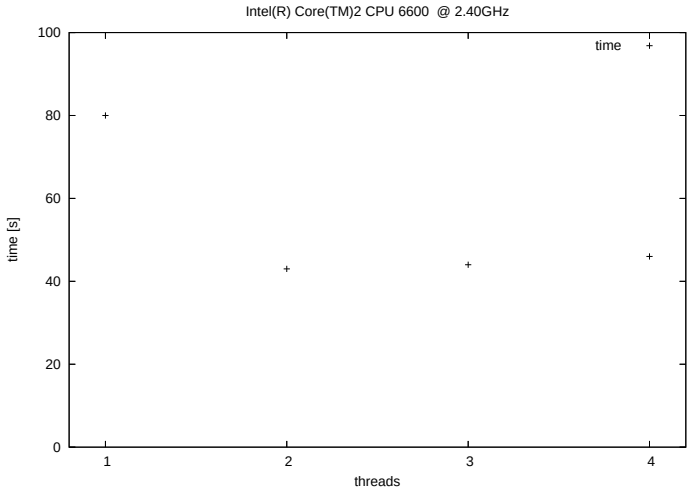
SHA(6.6(b)) jakožto svým způsobem referenční algoritmus ukazuje prakticky dvojnásobné zrychlení.

Shrnutí

Je vidět, že až na SGA byla paralelizace poměrně úspěšná. Evoluční techniky zajisté nepředstavují optimální řešení hledání koeficientů Fourierovy řady, během měření byl výsledek na mnoha místech odlišný od hledané funkce, která je z testovacích důvodů sama o sobě konečnou Fourierovou řadou. Přesto byla často cena velmi nízká.



(a) SOMA - hledání koeficientů Fourierovy řady (dvoujádrový procesor)



(b) SHA - hledání koeficientů Fourierovy řady (dvoujádrový procesor)

Obrázek 6.6: Zrychlení výpočtu koeficientů Fourierovy řady na dvoujádrovém procesoru(SOMA,SHA)

6.2 Zrychlení dosažené pomocí OpenMP čtyřjadrovém procesoru

Zrychlení bylo testováno na školním serveru merlin v době, kdy byl na tomto serveru minimální provoz. Merlin má dva dvoujádrové procesory `Dual Core Opteron 2216`, z nichž každý má následující parametry:

```
model name      : Dual-Core AMD Opteron(tm) Processor 2216
cpu MHz         : 1000.000
cache size      : 1024 KB
cpu cores       : 2
```

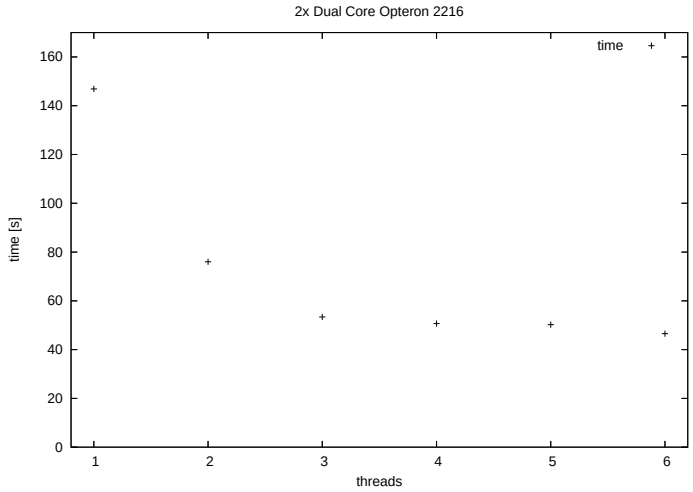
Parametry algoritmů jsou stejné jako u testů provedených pro dvoujádrový procesor, a jsou popsány v předchozí části textu.

6.2.1 Inverzní fraktální problém

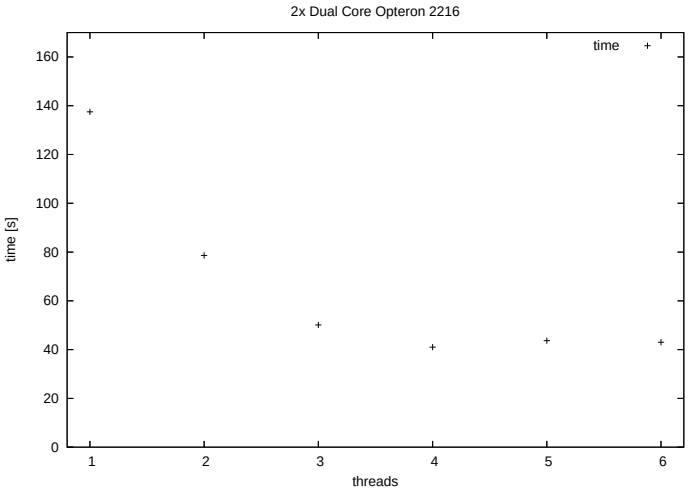
Jak u standardního genetického algoritmu (6.7(a)) tak i u diferenciální evoluce(6.7(b)) je možno pozorovat zrychlení, které však přímo neodpovídá počtu procesorů, ale postupně se s narůstajícím počtem procesů roste, až se zastaví u 4 procesů. U samoorganizačního migračního algoritmus(6.8(a)) je zrychlení dané počtem procesorů o něco stálejší, rozdíl mezi 3 a 4 procesy je výraznější. Naopak průběh doby stochastického horolezeckého algoritmu (6.8(b)) se výrazně neliší od prvních dvou grafů.

6.2.2 Učení umělé neuronové sítě

Standardní genetický algoritmus(6.9(a)) nezrychlil tak, jak měl. Z grafu je patrné, že zrychlení všelijak osciluje Naopak diferenciální evoluce (6.9(b)) a SOMA (6.10(a)) reagovaly poměrně dobře. Jen ten rozdíl mezi 3 a 4 procesy je poněkud malý. Je vidět, že režie na vedení procesů již začíná převažovat nad zrychlením. Je to dáno asi tím, že úloha nebyla



(a) SGA - hledání fraktálu (2xdvoujádrový procesor)



(b) DE - hledání fraktálu (2xdvoujádrový procesor)

Obrázek 6.7: Zrychlení výpočtu fraktálu pomocí OpenMP na 4-jádrovém procesoru(SGA,DE)

dostatečně výpočetně náročná na výpočet jednoho vlákna. Různé druhy plánování vláken by zde snad mohly pomoci. Stochastický horolezecký algoritmus 6.10(b) dopadl obdobně.

6.2.3 Hledání koeficientů Fourierovy řady

Standardní genetický algoritmus přinesl jisté zrychlení(6.11(a)), diferenciální evoluce (6.11(b)) rovněž. Samoorganizační migrační algoritmus také funguje - viz. 6.12(a) Stochastický horolezecký algoritmus je také určitým způsobem urychlen (6.12(b)).

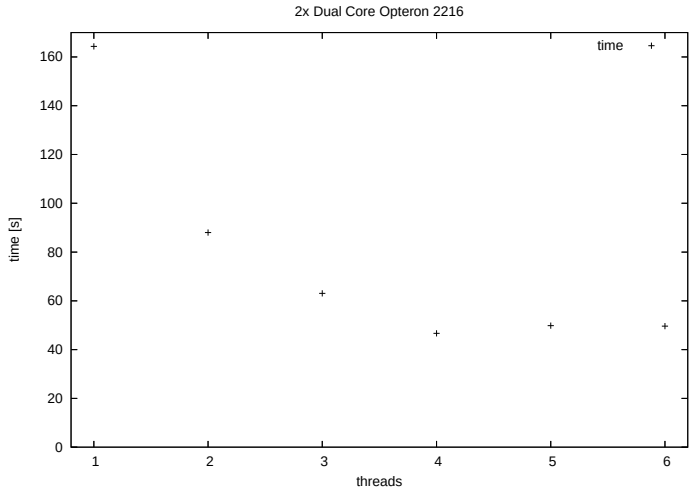
6.3 Zrychlení dosažené pomocí MPI

Bohužel se nepodařilo dosáhnout zrychlení pomocí metod, jenž byly nastíněny výše. I při rozjetí výpočtu na více stanicích trvá evoluce všech pořád stejně dlouho, a v případě, kdy je jedna dimenze rozdělena dojde dokonce ještě k většímu zpomalení. Jediným správným řešením by pravděpodobně bylo celou paralelizaci koncipovat jako jeden evoluční algoritmus, a jeho části rozdělit pomocí MPI mezi stanice na síti. Toto bohužel není možno realizovat jednak proto, že návrh knihovny k tomu není vhodný, a jednak z důvodů časových. Následující tabulka ukazuje, jaké časy byly získány paralelizací algoritmu SOMA pomocí

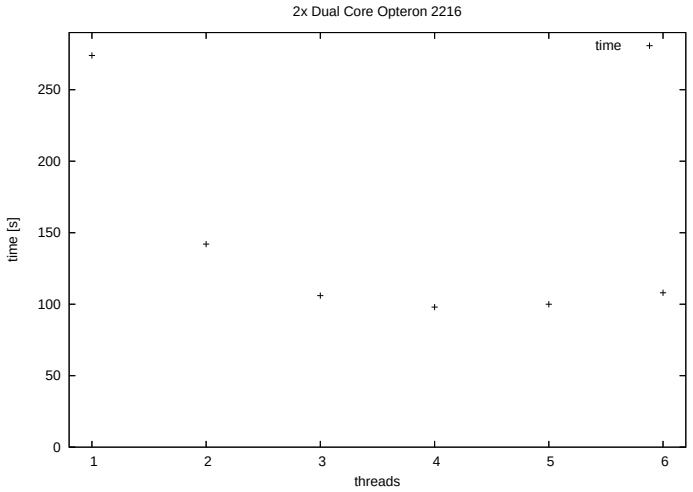
procesy	čas
1	56s
2	1m01s
3	1m04s
4	46s

MPI.

Z tabulky je patrné, že k nějakému zásadnímu urychlení nedochází, a případné zrychlení je spíše dílem náhody, kdy některý z algoritmů na některé ze stanic náhodou dospěje k lepšímu řešení. Výměna částí populace možná může v některých případech přispět k větší diverzibilitě populace, a tudíž k vyváznutí z lokálního extrému. To jsou asi jediné přínosy, které touto implementací byly dosaženy.



(a) SOMA - hledání fraktálu (2xdvoujádrový procesor)

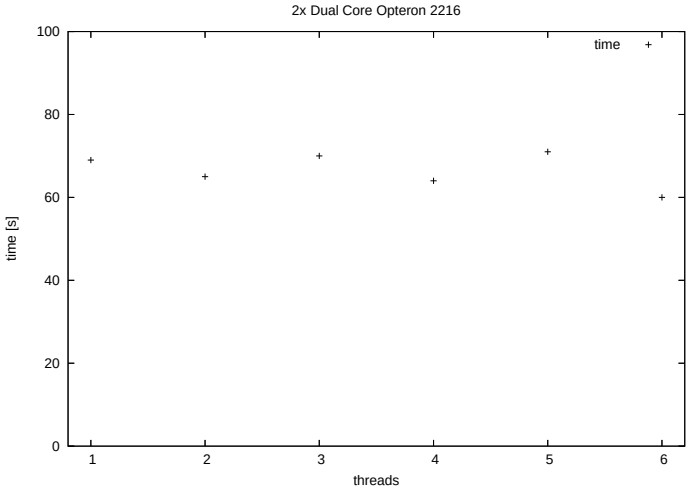


(b) SHA - hledání fraktálu (2xdvoujádrový procesor)

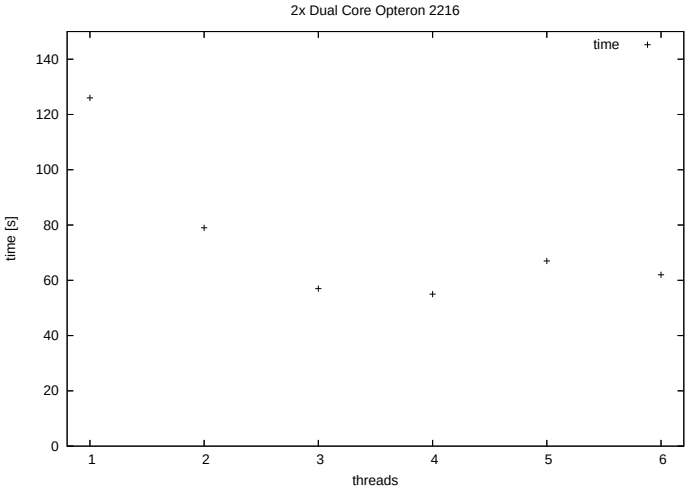
Obrázek 6.8: Zrychlení výpočtu fraktálu pomocí OpenMP na 4-jádrovém procesoru(SOMA,SHA)

6.4 Zrychlení dosažené kombinací MPI a OpenMP

Přirozeně, že pokud funguje OpenMP, tak dojde k urychlení na stanicích, jenž tohoto dovedou využít. Pokud všechny stanice mají stejný počet vláken, dá se očekávat, že celková doba výpočtu se zkrátí, někdy až téměř o polovinu, což je ideální případ. Jinak platí stejný závěr jaký byl napsán výše - protože přínos MPI se dostatečně neprojevil, není urychlení prakticky poznat. Přesto existuje jistá naděje, že existuje úloha, kde zrychlení bude přínosné.

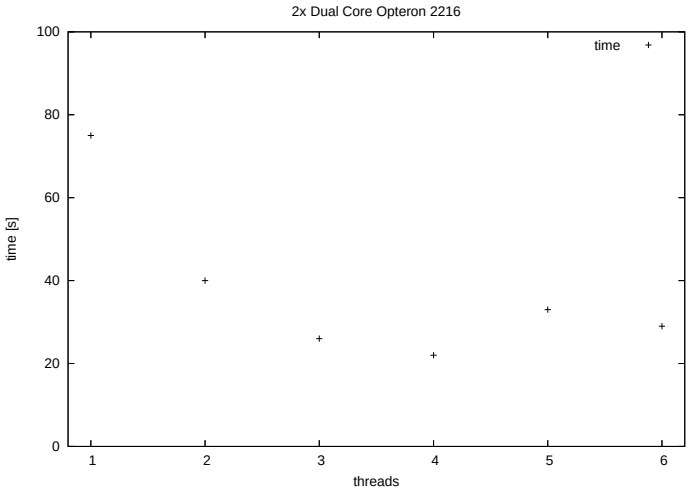


(a) SGA - hledání vah neuronů (2xdvoujádrový procesor)

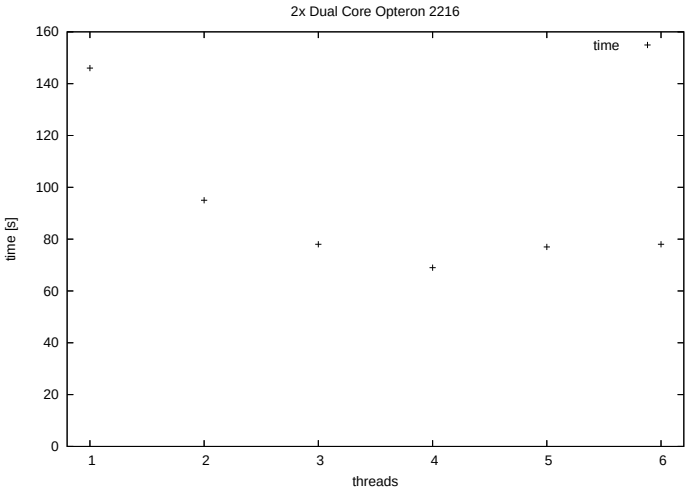


(b) DE - hledání vah neuronů (2xdvoujádrový procesor)

Obrázek 6.9: Zrychlení výpočtu neuronové sítě pomocí OpenMP na 4-jádrovém procesoru(SGA,DE)

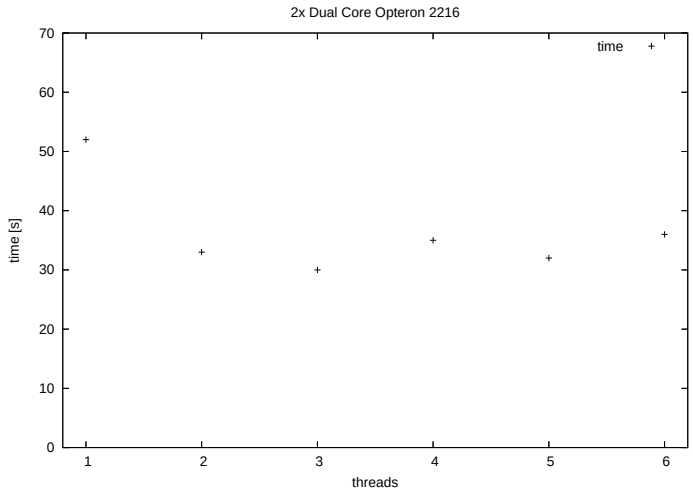


(a) SOMA - hledání vah neuronů(2xdvoujádrový procesor)

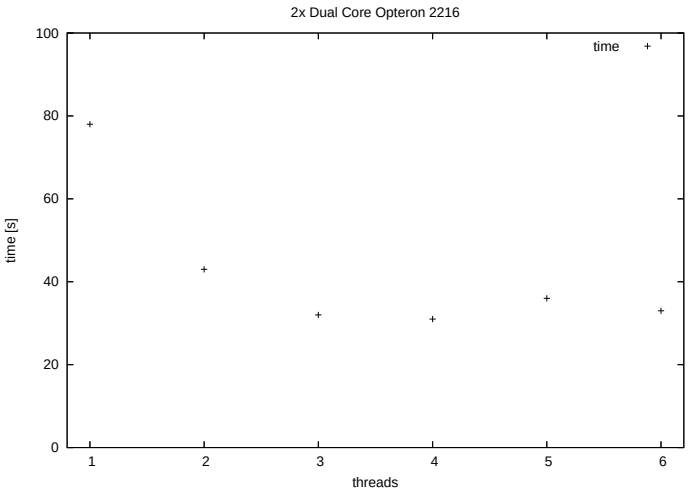


(b) SHA - hledání vah neuronů (2xdvoujádrový procesor)

Obrázek 6.10: Zrychlení výpočtu neuronové sítě pomocí OpenMP na 4-jádrovém procesoru(SOMA,SHA)

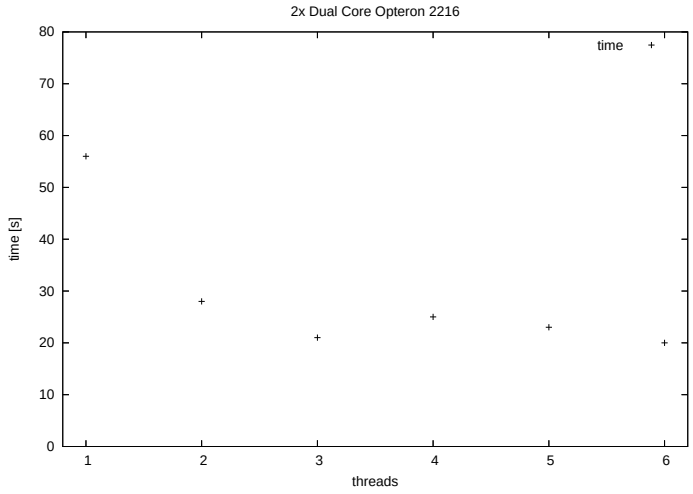


(a) SGA - hledání hledání koeficientů Fourierovy řady (2xdvoujádrový procesor)

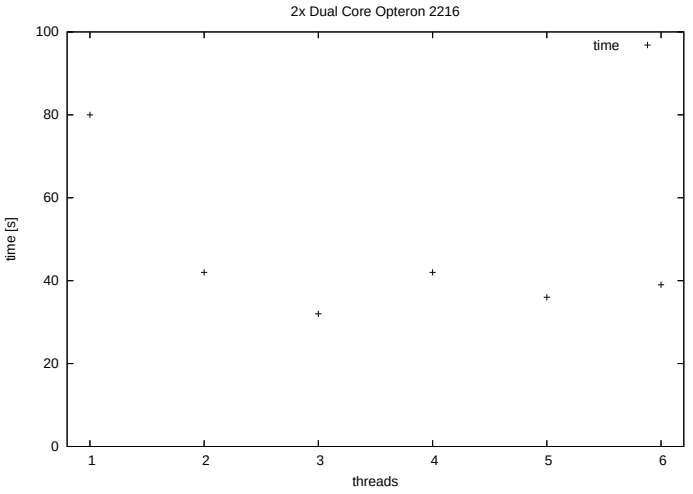


(b) DE - hledání hledání koeficientů Fourierovy řady (2xdvoujádrový procesor)

Obrázek 6.11: Zrychlení výpočtu koeficientů Fourierovy řady pomocí OpenMP)



(a) SOMA - hledání koeficientů Fourierovy řady (2xdvoujádrový procesor)



(b) SHA - hledání hledání koeficientů Fourierovy řady (2xdvoujádrový procesor)

Obrázek 6.12: Zrychlení výpočtu koeficientů Fourierovy řady pomocí OpenMP

Kapitola 7

Závěr

V rámci práce bylo prostudována paralelizace evolučních algoritmů. Bylo dosaženo zrychlení pomocí OpenMP na vícejádrových systémech, a bylo experimentováno s urychlením za pomoci MPI. Jako přínos práce lze uvést vytvoření objektově orientované knihovny, byť její kód není vždy zcela nejčistší. Rovněž byla provedena důkladná měření, a uvedena ta, která vedla k prokazatelnému zrychlení.

Jako možné pokračování práce si lze představit dokončení paralelizace pomocí MPI takovým způsobem, že se nebude uvažovat více oddělených populací vyvíjejících se nezávisle, což se ukázalo jako řešení nevedoucí ke vždy prokazatelnému urychlení, ale provede se rozložení náročných částí algoritmů mezi procesy MPI. Důvodem, proč se paralelizace pomocí MPI nepovedla, byl mylný předpoklad, že výměna několika jedinců povede k výrazně rychlejší konvergenci. Skutečnost se ale ukázala být taková, že pro paralelizaci by asi bylo mnohem vhodnější použít hrubou výpočetní sílu.

Dalším možným návrhem budoucího zlepšení je pročištění návrhu knihovny, a přistoupení k návrhovým vzorům. Návrh knihovny byl inspirován snahou přiblížit se návrhovým vzorům, ale kvůli ne příliš velkým zkušenostem s návrhovými vzory došlo k návrhu, který se sice na první pohled může zdát přehledným, ale autor vnitřně cítí, že by bylo možno dosáhnout jak mnohem čistšího návrhu, tak implementace. V podstatě se zde nabízí implementace nové knihovny, která bude mnohem čistší, rozsáhlejší a více funkční.

Evoluční techniky, to není jen hledání extrémů nějaké funkce. Jednou z oblastí, kde je jich využito, je tzv. genetické programování, kdy jsou jedinci reprezentování např. pomocí nějakého stromu, či jiné blíže nespecifikované datové struktury. Jako další možné pokračování této práce se proto jeví paralelizace genetického programování.

Literatura

[1] BLAISE, B.: Message Passing Interface (MPI). [Online], [rev. 2007-09-13], [cit. 2008-05-12].
URL https://computing.llnl.gov/tutorials/mpi/

[2] DVOŘÁK, V.: *Architektura a programování paralelních systémů*. Vysoké učení technické v Brně, 1994, ISBN 80-214-2608-X.

[3] MAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽANSKÝ, J.; aj.: *Umělá inteligence (1)*. Academia Praha, 1993, ISBN 80-200-0496-3.

[4] MAŘÍK, V.; ŠTĚPÁNKOVÁ, O.; LAŽANSKÝ, J.; aj.: *Umělá inteligence (3)*. Academia Praha, 2001, ISBN 80-200-0472-6.

[5] STORN, R.: Differential Evolution Homepage. [Online], [rev. 2007-08-19], [cit. 2008-05-12].
URL http://www.icsi.berkeley.edu/~storn/code.html

[6] ZELINKA, I.: SOMA Homepage. [Online], [rev. 2004-08-03], [cit. 2008-05-12].
URL http://www.ft.utb.cz/people/zelinka/soma/

[7] ZELINKA, I.: *Umělá inteligence v problémech globální optimalizace*. Nakladatelství BEN - technická literatura, 2002, ISBN 80-7300-069-5.